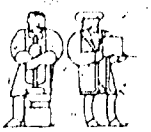
LABORATORY FOR COMPUTER SCIENCE



MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TR-446

MANAGING PARALLELISM AND RESOURCES IN SCIENTIFIC DATAFLOW PROGRAMS

David E. Culler

March 1990

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-446	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-84-K-0099
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Boulevard Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Managing Parallelism and Resources in Scientific Dataflow Programs

12. PERSONAL AUTHOR(S)
David E. Culler

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	March, 1990	242

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Bounded loops, Controlling parallelism, Dataflow, Parallelism, Program analysis, Resource management, Throttling

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Exploiting parallelism to achieve high performance invariably increases the resource requirements of a program. This is particularly serious under dynamic dataflow execution, because all the potential parallelism in a program is exposed. The resource requirements can be excessive, often leading to deadlock. This phenomenon is documented using parallelism and resource profiles derived under an ideal dataflow execution model.

The thesis examines how resource requirements can be managed effectively by controlling the ways in which parallelism is exposed. A mechanism for controlling parallelism in scientific programs, called k-bounded loops, is presented. This involves compiling loops into dataflow graphs in a manner that allows the maximum number of concurrent iterations to be set dynamically, when the loop is invoked. A policy for employing this mechanism is developed and

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Publications Coordinator	22b. TELEPHONE (Include Area Code) (617) 253-5894 22c. OFFICE SYMBOL

☆U.S. Government Printing Office: 1985—607-047

19. (cont.)

tested on a variety of programs. Through static analysis of the program, parametric *resource expressions* are formulated and the potential parallelism is characterized. Based on this analysis, the program is augmented with resource management code that computes the k-bounds by simple formulae, involving program variables and an overall resource parameter that reflects the capacity of the machine. This approach is shown to be effective for containing the resource requirements of scientific dataflow programs, while exposing adequate parallelism.

Managing Parallelism and Resources in Scientific Dataflow Programs

David E. Culler

MIT / LCS / TR-446
November 1989

This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099. The author also received support from the Unisys Corporation as a Burroughs Scholar.

This report was originally published as the author's doctoral dissertation (MIT, Department of Electrical Engineering and Computer Science, June 1989).

Managing Parallelism and Resources in Scientific Dataflow Programs

David E. Culler

Technical Report MIT / LCS / TR-446
November 1989

MIT Laboratory for Computer Science
545 Technology Square
Cambridge MA 02139

Abstract

Exploiting parallelism to achieve high performance invariably increases the resource requirements of a program. This is particularly serious under dynamic dataflow execution, because all the potential parallelism in a program is exposed. The resource requirements can be excessive, often leading to deadlock. This phenomenon is documented using parallelism and resource profiles derived under an ideal dataflow execution model.

The thesis examines how resource requirements can be managed effectively by controlling the ways in which parallelism is exposed. A mechanism for controlling parallelism in scientific programs, called k-bounded loops, is presented. This involves compiling loops into dataflow graphs in a manner that allows the maximum number of concurrent iterations to be set dynamically, when the loop is invoked. A policy for employing this mechanism is developed and tested on a variety of programs. Through static analysis of the program, parametric *resource expressions* are formulated and the potential parallelism is characterized. Based on this analysis, the program is augmented with resource management code that computes the k-bounds by simple formulae, involving program variables and an overall resource parameter that reflects the capacity of the machine. This approach is shown to be effective for containing the resource requirements of scientific dataflow programs, while exposing adequate parallelism.

Key Words and Phrases: Bounded loops, Controlling parallelism, Dataflow, Parallelism, Program analysis, Resource management, Throttling

Acknowledgments

I am privileged to consider myself a second-generation computer scientist, and I thank my father, Dr. Glen Culler, for starting me down this path. I often recall his words on the day that I went to work for him, "The first thing you will need to learn in this business is how to throw out everything you've done and start over." Fortunately, what we throw out gets better and better. I am grateful to my advisor and friend, Prof. Arvind, for having the courage to "start over" and re-evaluate the foundations of computer architecture in order to make sense of parallel computing. That vision inspired me to come to MIT, as it did the collection of fine individuals that have made the Computation Structures Group so exciting and so fulfilling. Special thanks go to Greg Papadopoulos, my "partner in crime." Together we have grasped what we could not reach alone. I am looking forward to the great things we do together in the future. Thanks to Ken Traub, the man with the most orderly mind it has ever been my pleasure to know. Your compilation work has been essential to our progress as a group, and this thesis could not have been done without it. Thanks also to Bob Iannucci, the empire builder. Your vision and diligence was the force behind much of the research environment we take for granted in CSG. Steve Heller, we are all indebted to you for the humanness that you brought to the group. I aspire to learn new jokes that merit a position on your whiteboard. Bhaskar Guha Roy, my office-mate, colleague, and dearest friend, it is with great sadness that I express my gratitude for all that you brought to my life. I always assumed we would journey forward together. Thanks to Andy, Richard, Keshav, Vinod, Steven, Natalie, Nikhil and all the rest of the members of the Computation Structures Group for making this special place, what it is.

I am grateful to my excellent committee members, Prof. Robert Halstead and Prof. Rishiyur Nikhil, for their valuable contributions and to my unofficial readers, Dr. Kenneth Traub and Ms. Natalie Tarbet. Prof. Michael Dertouzos, $\epsilon\acute{\upsilon}\chi\alpha\rho\iota\sigma\tau\tilde{\omega}$ for your support and encouragement.

Thanks to my New England friends, including Marcela, C.K., Precila, Joo Kun, Deborah, Josef, Gita, Elizabeth, and the little Papadopouli, and to my old California ones, Jamie, John, Catherine, Alan, Pat, Noreen and Michael, for keeping life fun. With the deepest love and gratitude, I thank my parents, brothers, and sister for their love, kindness, and unending support. Without you, none of this would have been possible. Also, I am grateful to my other family, the Mayenos, for welcoming me into their hearts and their home.

Most of all, I thank Sara Mayeno, my wife, friend, partner, and love. Your love and support were always there, keeping me going; your patience made studying at MIT possible. You've brought into the world the most marvelous child, our son Ethan, and nurtured us with your love. Words will never express my gratitude. Sweetheart, we're going home!

It is May in Boston,
and the fruit trees are in bloom.
Petals fall gently to the ground.
I gather them in my hands and
feel their softness against my face.
Memories are like that.

In memory of Bhaskar Guha Roy

Contents

List of Figures

Chapter 1

Introduction

One of the facts of life for high-performance computing is that more parallelism invariably means more storage. We can observe this phenomenon on conventional machines to a limited extent, but for dataflow machines it presents a very serious problem. The dynamic dataflow approach exposes and exploits parallelism at *all* levels, regardless of program structure. As a result, ample parallelism is available on a broad class of programs, however, the resource requirements of many programs are tremendous. This thesis explores how the resource requirements of dataflow programs can be maintained at reasonable levels by controlling the ways in which parallelism is exposed. A mechanism for controlling parallelism is presented and a policy for employing the mechanism is developed in the context of scientific applications. This mechanism is called k-bounded loops.

To gain perspective on the problem at hand, consider some of the traditional techniques for exposing parallelism in programs. Starting at the lowest level, pipelined instruction execution allows parallelism to be exploited within a sequence of scalar instructions. In order to benefit from this, the compiler breaks a basic block into a collection of independent computational threads and then attempts to weave these together so as to maximize the distance between the point where a register value is defined and the point where it is first used[43, 51]. This requires that registers are available to support each of the concurrent threads, thus motivating architects of such machines to enlarge the register set[50, 75]. Not surprisingly, these compilation techniques stand in direct contrast to traditional techniques for minimizing register usage, which attempt to place all uses of a register value as close as possible to the point of definition[2].

An important technique for increasing the number of independent threads is *scalar renaming*, which replaces multiple updates of a single variable by definitions of multiple distinct

variables[1]. Thus, parallelism is exposed at a cost of increased number of local variables. A related technique, *vector expansion*, is crucial for exposing vector operations within innermost loops[64]. Scalar variables defined within a loop, including the temporaries implicit in arithmetic expressions, are expanded into arrays of dimension equal to the number of iterations of the loop. Here we begin to see significant resource expansion due to exposing parallelism.

Various techniques have been developed for exploiting outer loop parallelism on multiprocessors[32, 34, 61]. Basically, a program is represented by p concurrent tasks, one per processor, operating out of a common global memory. These can cooperate in performing a parallel loop by each performing a subset of the iterations[76]. At the very least, the amount of storage used for the local variables of the loop is multiplied p-fold. If the amount of local variable storage is large, *e.g.*, if each iteration involves vector operations with vector temporaries[64], the total storage increase can be dramatic.

More generally, if a dynamic form of task creation is employed[60, 82], such as fork or parallel-call, the number of concurrently active tasks, and hence the total amount of task state, can increase *exponentially* over an equivalent program employing only sequential calls. It does not matter whether the fork is explicit or implicit: if it is treated as a directive to perform two tasks in parallel, a binary-recursive program expands into a tree of active tasks, while the equivalent sequential program has only a single path down this tree active at any time — the call stack.

Even the traditional techniques for exploiting asynchronous I/O increase storage requirements. Multibuffering requires that separate storage areas be provided for the working data and data being transferred to or from secondary storage. Multiprogramming avoids the complexity of buffered I/O on a single task, at the cost of maintaining several concurrent working sets.

To appreciate the severity of the resource problem in a dynamic dataflow context, imagine employing all these conventional techniques simultaneously. Exposing parallelism at several levels has a multiplicative effect on parallelism, but also on resource requirements. This might appear to support the case for more traditional parallel architectures that support only specific forms of parallelism, *e.g.*, only vector parallelism or only outer loop parallelism. However, if only a particular form of parallelism is exploited, the "Amdahl's Law" phenomenon manifests itself, since large-scale applications often exhibit a variety of different forms of parallelism [58]. Accelerating only a portion of the program makes the remaining portion more critical. Thus,

it is desirable to be able to exploit *all* forms of parallelism, but also to be able to control the amount of parallelism, so the entire machine is kept busy, while resource requirements remain reasonable.

The parallelism/resource problem can now be stated more precisely. In general, a parallel program defines, for a given input, a partial order on the execution of its constituent operations. (This partial order is manifest in a dataflow graph program representation, but present to some degree in other representations as well[40, 55].) Different execution schedules exhibit different amounts of parallelism with different resource requirements; more of the first generally implies more of the second. Ideally, we want to arrive at a schedule or execution strategy that exposes enough parallelism to keep our machine completely utilized, while remaining within its resource capacity. If the execution schedule is too aggressively parallel, the program may deadlock for lack of resources when a more conservative execution strategy would succeed. This hazard has lead a number of researchers to consider what might be called *resource-limited data-driven* execution[26, 78, 79]. The goal is to allow only as much parallelism as can be exposed within a given resource budget.

In this thesis, we explore a particular form of resource-limited dataflow execution in the context of scientific applications, *i.e.*, programs dominated by iterative computations over large regular data structures with known lifetimes. The basic idea is quite simple — loops are compiled in a manner such that the maximum number of concurrent iterations can be set at the time the loop is invoked. We call this approach k-bounded loops, as a dynamically determined unfolding parameter, k, is associated with each loop[6, 27]. The k-bounded loop approach has many virtues; it facilitates use of fixed-size tags in tagged-token dataflow architectures[7, 45, 52], provides a natural mechanism for recycling resources within the dataflow graph, and makes attractive a new class of dataflow architectures which avoid associative matching through an explicit token store mechanism[74]. The k-bounded loop approach also permits a great deal of flexibility, as the parallelism exposed in a program can be adjusted dynamically with each loop invocation. However, with this flexibility comes the necessity of determining how each of the loop parameters should be set for a given program. Usually the best setting will depend on the machine configuration and the program inputs. We want to know whether it is possible to do an adequate job of establishing these parameters automatically. This is the primary focus of the thesis: developing a policy for determining loop k-bounds in large-scale scientific applications.

The scope of this work is intentionally restricted to a specific class of programs under a dataflow execution model. Thus, the particular mechanism developed for controlling loop unfolding is specific to dynamic dataflow architectures and the issues we face are affected by the language *Id*. Nonetheless, one may expect that a general understanding of the behavior of bounded loop programs may well apply to more traditional execution models, at least as parallelism is aggressively exploited in conventional programs[30, 65, 76].

Our focus is on scientific programs, so we restrict our attention to programs employing only iteration and primitive recursion[66]. We must assume, however, that the program involves multiple, diverse phases, each with potentially heterogeneous computations, and potentially overlapping in some fashion. Thus, rather than optimal scheduling for little kernels, the goal is to control program unfolding in the large. The resources of concern are local data storage, *i.e.*, tokens or activation frames, and data structure storage. Each concurrent iteration offers a certain amount of parallelism, but places a claim on some amount of each resource. We will compile these programs in such a way that the program *unfolds* until it makes use of the resources allocated to it, and then simply recycles resources within the graph until it completes.

1.1 Problem Definition

Our goal is to show that k-bounding loops is a viable means of managing parallelism to control resource usage in scientific programs. The mechanism is clearly effective in controlling the behavior of any loop in isolation[27], and it has had a strong influence on the evolution of dataflow architectures[74]. Thus, in essence we must demonstrate a largely automatic technique for assigning the k-bound parameters in real programs. Basically, we require that the k-bound assignment prevent deadlock due to excessive resource usage, while exposing enough parallelism to keep the machine busy. Thus, the ultimate goal of this work is the following.

> Given a program and a machine configuration, *i.e.*, storage capacity and number of processors, provide a policy for assigning loop bounds, that keeps resource usage within the capacity of the machine and exhibits adequate performance, say 75% utilization of all processors.

Unfortunately, like many "systems" problems, this really involves a variety of interrelated problems. Clearly, it requires that resource usage be sufficiently contained that the program

does not deadlock. However, the performance and utilization delivered will also depend on a host of lower level factors, such as how work is distributed over the machine, the scheduling policies employed, the happenstance of network collisions, etc. Thus, this goal is really too vague and involved to attack on one go. In this work, we want to focus on the question of how program unfolding can be controlled so that adequate parallelism is exposed within certain resource limits. To this end, we need to refine the problem and isolate the parallelism/resource issue.

1.1.1 Execution model

A major point of difficulty in addressing the pragmatic goal above has to do with modeling the behavior of real dataflow machines, or classes of such machines. The highly asynchronous nature of this sort of machine complicates characterizing its execution behavior. Moreover, the architectures themselves are evolving rapidly[55, 74]. In any case, the details of the machine distract from the real problem, namely, controlling the behavior of programs. The primary issue is the resource constraint, *i.e.*, deadlock avoidance, while performance is the secondary concern. Thus, it makes sense to trim away the awkward set of issues involved with realistic machine configurations and performance characteristics. Instead, we focus on program behavior under an idealized execution model, that assumes infinite processors, unit time operation, and instantaneous communication. This model is well defined for dataflow programs and reflects the inherent behavior of the program itself.

Since a dataflow program defines a partial order on the scheduling of its constituent operations, there is tremendous latitude in how operations are scheduled on any particular dataflow machine. Therefore, given a program and a resource limit, we must provide a policy for assigning loop bounds that honors the resource limit under *all* legal execution schedules. However, we will judge the quality of the policy against the ideal model. Our goal is to produce a policy which, for each value of the resource limit, achieves the minimum possible completion time under the ideal execution model, *i.e.*, minimum critical path length, within the resource limit. Since the total number of operations is nearly independent of the k-bound assignment, at least for reasonably good assignments, we can view minimizing the critical path and maximizing the average parallelism as synonymous. An assignment that achieves near-optimal performance under the ideal model is likely to demonstrate adequate parallelism on a real machine, if any

assignment would. This thesis strives to develop good policies in this abstract context, leaving as future work the investigation of how these perform on real machines.

1.1.2 Policy

We still must refine the notion of policy. At the very least, a k-bounding policy embodies an algorithm for assigning loop bounds. The discussion above gives a specification for the algorithm as a kind of optimization problem: given resource constraint r, determine a k-bound assignment that maximizes parallelism within this constraint. So why not go solve this optimization problem? Part of the answer is that the optimization problem is very poorly defined unless we actually run the program to extract certain parameters, such as the size of the data structures and the iteration counts for the loops. The cost function (resource usage) is non-linear and may be dependent on input values to the program and even values computed within the program. Even the space to optimize over and the objective function are highly data dependent. Really, we are looking for solutions to a class of optimization problems, because we require that programs be configuration independent: a single program should suffice for any reasonable problem size on any machine configuration. We do not accept recompiling the program for each problem size and particular machine configuration. Thus, much of the assignment must be determined dynamically during program execution. As such, the assignment must be determined very quickly. We cannot afford to search or solve general constraint systems, but must compute k-bounds according to simple rules. These rules embed higher-level policies, such as "favor unfolding of this loop over that one," which are derived through analysis of the program.

We can distinguish classes of policies by when the k-bound assignment is determined. At one extreme are *static* policies, which fix the loop bound for every loop, or possibly every loop invocation before the program starts execution. This is generally the approach taken in compiling FORTRAN programs for parallel machines. It requires that the size of the problem, iteration counts, and the size of the machine be known at compile time. We do not want to require such restrictions, although when the problem size is static we will exploit the opportunity to do more detailed analysis at compile time and to reduce the amount of dynamic decision making.

At the other extreme are *dynamic* policies, which use run-time machine status indicators, *e.g.*, the level of parallel activity and the availability of resources, to control program unfolding.

Purely dynamic policies are very general, but are not very robust. The problem is that being too eager at one point in the program may cause it to deadlock later on. In general, a huge number of decisions are made, most of which are insignificant while few are crucial. Without program analysis it is hard to focus on the significant decisions. In addition, the status of the machine may be a poor indicator of future loads, causing the decision mechanism to make incorrect choices.

Our approach falls between these extremes; the program is augmented at compile-time with code that determines the k-bounds dynamically based on some set of resource limits, program input values, and values computed within the program. It does not, however, take into account dynamic machine status information. A conceptual view of the process is illustrated in Figure 1.1. A program has a certain inherent behavior that is "bad" for a realistic machine, because of inordinate resource requirements and possibly excess parallelism. We augment this program to provide a new input, the resource limit r, and code for determining the various loop bounds based on this limit. The additional code essentially restricts the partial order so that the remaining legal execution schedules stay within the parameterized resource limit. This augmented program, when supplied with limit r, has an inherent behavior that is reasonably "good" for machines of a size related to r. Lower-level resource management problems, *e.g.*, assignment of work and data to processors, scheduling, etc., can be addressed in the context of this reasonably well-behaved program. This approach is applicable only to a restricted class of programs, but is quite robust.

1.1.3 Refined problem

Based on this discussion, our problem can be restated as follows:

> Given a program, provide rules for computing loop bounds, parameterized by resource constraints and certain program input values, that obtain close to minimum critical path length within the resource constraints.

Let us loosely define a *k-configuration* for a program as a setting of the k-bound of each loop invoked during the execution of the program. We will need a means of quantifying the worst-case resource load for a k-configuration under any execution schedule. The schedule followed by a realistic machine will certainly differ from our ideal model, but this change should not

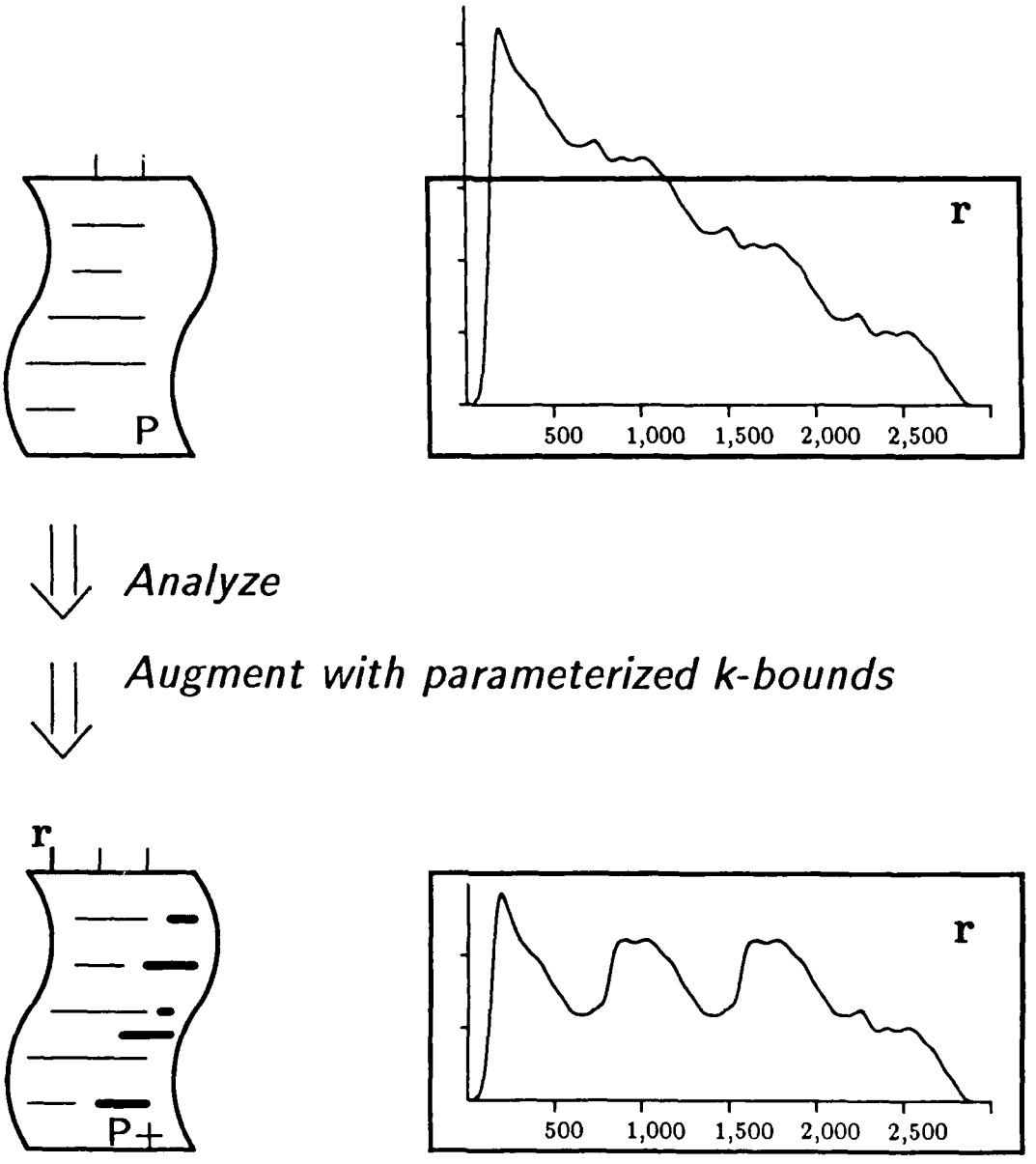


Figure 1.1: Conceptual View of Our Approach

violate the resource constraint. Failure to establish a tight bound on the resource load is one reason our policy may be sub-optimal, but also the execution schedule followed by a particular machine may not exhibit the worst-case resource load. We will also need to characterize, at least qualitatively, the behavior of a k-configuration in terms of average parallelism and overall execution cost. Together, these give an indication of the "goodness" of regions of the immense space of possible k-configurations.

Since we are interested in large-scale scientific applications, the policy must be derived with tractable compilation techniques and implemented with reasonable efficiency. We will end up with a policy for assigning loop bounds that is sensitive to some relatively small set of program characteristics. The backbone of our analysis is the *static invocation tree*, which gives a crude characterization of the relationships between the various loops in the program. It is built under two basic rules of composition, nesting and adjacency. We start by developing a simple behavioral model of bounded loops, and then consider nested loops of various sorts. Having fixed a set of guidelines for dealing with nesting, we consider adjacency. Conditionals are dealt with only briefly. Finally, we apply the rules developed and tested on small programs to a large program.

1.2 Overview of the Thesis

In the remainder of this chapter, we place this work in the context of related research in the field. In Chapter 2, we define an ideal dataflow execution model and identify the resources associated with the execution of a dataflow program. This gives rise to a characterization of programs by their ideal parallelism and resource profiles and provides a means to substantiate the claim that dataflow programs exhibit ample parallelism, but inordinate resource requirements. We define a more realistic execution model and show that restricting the processing power does not improve the resource situation. However, by controlling program unfolding, the resource requirement can be reduced without increasing execution time under the more realistic model.

In Chapter 3, we develop the basic mechanism for k-bounded loops and explore virtues of the k-bounded loop approach. In particular, it provides a natural means of recycling a variety of resources within the program itself, including data structures. Also, it suggests a path of architectural evolution toward radically simpler dataflow machines.

In Chapter 4, we formalize the notion of a loop bounding policy and lay a foundation for policy development. We develop a simple behavioral model of unbounded and k-bounded loops, *i.e.*, what parallelism, resource requirements, and execution costs they exhibit. This gives rise to the concept of *useful unfolding*, and our first important loop classification — loops with limited useful unfolding. These tend to be inner loops and loops with very strong recurrences.

In Chapter 5, we take up a study of simple nested loops. From this we articulate trade-offs in outer and inner loop unfolding, and demonstrate situations in which outer loop unfolding is superior. We also observe that certain regions of the space of k-bounds are particularly unattractive.

In Chapter 6, we examine the interplay of local storage limits and structure storage limits. Here we see situations in which inner loop unfolding is clearly superior. This gives rise to a simple classification scheme. We can characterize loops as: (1) having limited useful unfolding, (2) unattractive to unfold, and (3) attractive to unfold. In many cases, outer loops with deep dependencies that are hard to detect will be placed in the second category by virtue of large resource requirements, when they properly belong in the first. However, the weaker categorization is generally adequate to yield an effective policy.

In Chapter 7, we explore the problem of partitioning resources among adjacent loops. The basic outcome is a notion of proportional allocation, although certain special cases are shown to be important.

In Chapter 8, we bring these developments together into a coherent policy for assigning loop bounds and perform a case study on a fairly large program to validate the policy. One outcome of this study is the observation that detailed analysis is very important at the innermost level, which is where it is also most applicable. Deep dependencies at the outermost levels are also crucial, but difficult to catch through analysis. Fortunately, the programmer has the most intuition at that level and can offer guidance through simple annotations. At middle levels, where analysis is hard and the programmer is likely to have little intuition, we can generally do well with rather crude analysis.

Finally, in Chapter 9, we take a critical look at the basic k-bounded loop approach and examines its effectiveness as an instance of resource-limited data-driven execution.

1.3 Relationship to Parallel Fortran Execution

The work in exposing parallelism in sequential FORTRAN programs has traditionally been circumscribed by the kind of parallelism the potential target machines could exploit. For vector processors, the paramount goal is to expose inner loop **FORALL** constructs. In an MIMD context, there is somewhat more flexibility, but parallelism can only be exploited at a single loop level, in most cases. This must be an outer loop level in order to gain leverage over the sizable cost of dispatching parallel operations[33, 81]. It is generally assumed that the portions of the program that run in parallel are so large that the dispatch operation and even the rest of the program are negligible[33, 47, 48]. A few recent machines (such as the Cray-Xmp and Alliant FX-8[56]) encourage program transformations that expose multiple concurrent vector operations. For these transormations, the expansion of resource requirements is a significant concern.

Recent work in the parallel FORTRAN domain entertains scenarios more akin to the viewpoint taken here. Through aggressive renaming, anti-dependence can be essentially eliminated, leaving a representation of the partial order of data dependencies, much like a dataflow graph[30]. There are some subtleties as to what this means for data structures, but barring that point, one might use this framework to examine the inherent or asymptotic parallelism in a program. It provides a metric against which more conservative transformation policies can be gauged. Also, it provides a means for assessing the cost of parallelism in terms of resources required. Not surprisingly, the resource requirements can be tremendous in this scenario, much as they are for dataflow programs[65].

On the other hand, it is interesting to examine the traditional parallel FORTRAN work from the perspective developed here. In identifying a vectorizable loop of count n, we have effectively *exposed* n-fold parallelism. A pipelined vector processor can only obtain parallelism μ, determined by the depth of the functional unit pipeline and the latency to memory. Thus, in a simple pipelined model the asymptotic parallelism, as $n \rightarrow \infty$, is μ. Half of this value is obtained at $n = \mu$, and the increase is slow from there on[53, 67]. Usually, the performance determining factor in such a machine is the rate at which the sequence of memory addresses can be generated. Dividing this into the time to perform the fetch-operate-store gives μ. In dataflow loops, both the address generation and the operation are explicit, and we arrive at a similar formula for the useful unfolding of a loop and the obtained parallelism.

In exposing the vectorizable loop of count n, we may have introduced vector temporaries of length n through scalar expansion. By strip-mining[54] the loop into chunks of length k, we can reduce the size of the temporaries to k. In effect, we have reduced the exposed parallelism, bringing it closer to the amount that can be exploited. As we shall see for dataflow programs, the resource requirements are proportional to the amount of parallelism that is exposed, regardless of how much can be exploited. Historically, strip-mining was motivated by the presence of vector registers, but it has the benefit of simplifying the storage management problem in some regards. One can imagine compiling a program to run on a class of vector machines all with the same instruction set, but with differing memory capacity and pipeline depth. For small machines with shallow pipelines, we should adopt a small strip-mine value, whereas for larger machines with deeper pipelines, we should adopt a larger value. For one-dimensional vector operations of length n in the context of data structures of size $n \times n$, this may seem insignificant, but consider multiple vector processors that exploit parallelism at two levels or when parallel operation is possible at all levels, as in the dataflow setting. Then we see the multiplicative effect on resources.

There is a fairly strong relationship between the bounded loop execution explored here and Cytron's theoretical work in limited-processor scheduling of Doacross loops[29]. He considers execution of programs on an unbounded number of "virtual processors," where each loop iteration can be assigned to a distinct virtual processor. For nested loops, the iterations for each invocation of a loop are assigned to distinct collections of virtual processors, so parallelism at several loop levels is considered. Dependencies between iterations are resolved by inserting delays[28] so that an iteration does not access a value produced by another until that value is present. He avoids taking a position on whether this requires explicit synchronization operations or not.

Cytron's virtual processors correspond roughly to our *activation frames.* The model he develops for loop behavior is similar to what appears in Chapter 4, and a similar conclusion is reached, namely, that a given loop may have a natural limit on the number of concurrent iterations. He makes a substantial effort to avoid what we call partial-wave effects when limiting the number of processors. On the other hand, he "compiles in" the problem size and machine size, where we leave these as parameters. He is forced to define a precise schedule from the inside out, where we have more flexibility. He assumes the outer loop is the most advantageous

place to exploit parallelism, whereas we are fundamentally concerned with cases in which this is not so. We must address dynamic allocation, which Cytron does not consider, and deal with a completely asynchronous environment.

Finally, some of the work in parallel FORTRAN execution could be applied to further improve the behavior of dataflow programs. As the class of target parallel machines becomes more diverse, a general theory of loop interchange is emerging[4]. We will see that under our current approach to compiling to dataflow graphs, the parallelism available in innermost loops is somewhat limited. It can be improved by static loop unrolling, but might also profit from interchanges that move strong recurrences into the innermost loop. This may be particularly important when a very high-level programming style is adopted and the programmer has not been concerned with the organization of loop nests.

1.4 Related Work and Historical Perspective

Research in dealing with the parallelism/resource problem has a considerable history within the functional language community. In this section we examine briefly the relevant pieces of that work to establish a context for the results presented here.

1.4.1 Token-storage overflow

The concept of tagged-token dataflow architectures emerged during the late '70s at the University of California, Irvine[12, 14, 15] and the University of Manchester[46] after extensive work at M.I.T and elsewhere in static dataflow architectures[24, 35, 36, 37, 38, 62, 80]. It was clear that the most complex aspect of this model of computation was realizing the token matching function efficiently. A number of groups embarked on projects to understand how large this associative store had to be and how it could be implemented[5, 52, 83, 84]. The Manchester group developed a prototype machine, using a parallel hash table and overflow unit[45], while the group at M.I.T. developed a detailed simulator for the MIT Tagged-Token Dataflow Architecture[9, 16, 21]. The short answer to the question of adequate matching-store capacity was, "too big." As the language tools advanced to a point where meaningful programs could be compiled, it became clear that most problems exhibited very large matching-store requirements.

Key to understanding the matching-store requirements of programs is the concept of the

dynamic execution tree[26]. If we consider each invocation of a user-defined function and each iteration of a loop to be a "task," then the execution of a program defines a tree, where the nodes are the tasks created during execution and the children of a node are the tasks it created. Observe that multiple invocations within a task cause the tree to branch with fixed degree, whereas loops cause branching of degree determined by the number of iterations. This tree describes the entire execution history of a program on a given input; at any point during the execution only a subtree of it will exist. Under sequential execution, the active subtree is a single branch, described by the call stack. Under parallel execution the active subtree may encompass the entire tree! In regard to matching-store capacity, observe that waiting tokens serve a dual role; they correspond to registers where computation is active and serve as local variables throughout the interior of the active subtree. Thus, it is easy to see how certain program structures cause the matching store requirement to grow arbitrarily.

With waiting tokens viewed in this way, an interesting aspect of tagged-token dataflow architectures is that registers and local storage are not managed by the compiler, but are allocated dynamically by the hardware in an amount dependent on the particular execution schedule, which is determined by the happenstance of asynchronous events during program execution. The author's work[26] introduced a technique for computing the worst-case token storage requirement over all possible execution schedules, based on linear programming. This gave rise to the concept of bounded loops, as the linear program for a loop has no optimal solution precisely when the loop has potentially unbounded unfolding. It was shown that by adding auxiliary arcs to the dataflow graph, a loop could be forced to have bounded unfolding.

The way to reduce the matching store requirements of a program is to limit the breadth of the active subtree. In analogy with the technique for controlling loop unfolding, auxiliary arcs were proposed as a way to limit the static branching of the active subtree as well. The idea was to compile two versions of the graph, one that allowed the child tasks to execute in parallel and one that serialized them via auxiliary arcs. The version used for a particular invocation would be determined by status of the machine at the time of invocation, *i.e.*, level of parallel activity and availability of storage resources. This approach has several problems. First, it is not possible, in general, to serialize the execution of subtasks in a language with non-strict functions and data structures[86]. Second, even if a valid order can be determined for a meaningful class of programs, the policy for selecting which version to use is non-trivial[22]. Finally, the failure

mode is catastrophic. If scheduling is too eager when the tree is unfolding, the active subtree may consume so much of the resources that no single branch can extend far enough to reach a point of completion and make more resources available. Thus, the machine deadlocks. This can also occur when the selection criterion is fooled by a temporary period of low parallelism. Generally, a parallel program may exhibit phases of ample parallelism and constriction points where little parallelism is available. At a constriction point, there may be tasks that are ready to be invoked, but will only allocate resources and then wait for the critical dependencies of the constriction point to be resolved before making any real progress. A parallelism-based mode selection criterion may allow extensive branching during such periods and thereby arrive at the over-committed state discussed above. This phenomenon is much more likely under languages with non-strict functions and data structures.

1.4.2 Halstead's task queue

Halstead observed the parallelism/resource problem independently in the context of Multilisp[60]. While parallelism is explicit in Multilisp, in writing a program the programmer must take a position on whether a particular invocation or expression is to be evaluated in parallel or not. Halstead notes that for binary recursion under fair scheduling the parallel formulation can require exponentially more storage than the sequential formulation. This is again an execution tree argument. Sequential execution implements a depth-first traversal of the execution tree, while parallel execution tends toward a breadth-first traversal. His solution was to introduce a simple "unfair" bias toward depth in the definition of the parallel call.

The idea starts with the assumption that a processor executes a single task at a time, and that ready tasks are maintained in a system-wide task pool, represented by a collection of stacks[59]. Each processor maintains its own task queue, using a LIFO discipline, but an idle processor can "steal" tasks from other queues. The definition of the parallel call is such that the child continues execution on the processor that executed the call, while the parent that made the call is pushed back onto the pending task queue. Thus, if no processors are idle, the parent is suspended and, hence, cannot spawn additional children. Meanwhile, the child may be superseded by its children and execution tends toward a depth-first traversal. If a processor becomes idle, it takes the task at the top of its queue or steals a pending task from another queue, if its queue is empty. Thus, an idle processor may steal the suspended parent and resume

it while the child is executing.

For a single loop, perhaps represented as tail-recursion, that invokes some simple function, we can see how this approach might achieve the bounded loop behavior we desire, with one concurrent iteration per processor. The loop task spawns the function and is suspended until some idle processor reawakens it. The function for the next iteration is then spawned on that processor and the loop task is again suspended. This continues until a function invocation is spawned on each processor.

However, in practice the situation is more complicated. Calls are not the only reason that a task may suspend. With non-strict data structures, which can be implemented by futures[60], producers and consumers interact across different branches of the execution tree. Thus, the child may suspend temporarily and reawaken the parent. The parent may then spawn additional children, which may allocate resources even though the data they require have not been computed. This can give rise to the over-reaction in periods of low parallelism discussed above. Moreover, once the the depth-first ordering of the task queue is disturbed, a critical task may get buried in the queue and thereby cause the mechanism to break down. The child that was temporarily suspended may not be re-enabled until the parent has spawned off all its other children, and their children, and so on. The implementation of the task pool as a collection of local queues, potentially undermines the delicate ordering of suspended tasks. However, it also reduces the likelihood of a critical task remaining buried indefinitely. In summary, while the bias in the parallel call is an interesting mechanism, it has subtle, incompletely understood effects. In particular, it is unclear how effectively it contains the resource requirements of large programs in the presence of non-strict data structures. It makes no bottom-line resource guarantee.

1.4.3 Token queue techniques

The dataflow group at Manchester University explored a variety of techniques for controlling parallelism by modifying the low-level instruction scheduling mechanism. One approach was to maintain separate queues for instructions with one input and for those with two. Under the assumption that instructions have at most two outputs, the first category can never decrease the number of tokens in existence, while the second can never increase it. Instructions are scheduled from one or the other queue based on some measure of the activity level. This

approach does reduce the token storage requirements, but does not overcome the fundamental tendency toward breadth-first unfolding of the execution tree[78].

A second approach stemmed from the observation that a token stack, as opposed to a queue, gives preference to newly created activities. For recursive constructs this introduces a bias toward depth, something like Halstead's parental suspension technique. Unfortunately, for loops it tends to allow all the iterations to come into existence concurrently before any of them perform any useful work. This dichotomy motivated Sargeant's *Intelligent Token Queue*[79]. The idea was to switch the scheduling discipline dynamically, so that it would be stack based for recursive constructs and queue based for iterative ones. To reduce the complexity somewhat, switching was to be performed at the task level, rather than between individual instructions. This met with little success in practice[78].

1.4.4 Ruggiero's coarse-grained throttle

Ruggiero rejected these fine-grained mechanisms in favor of one that operates at the execution tree level[78]. The idea is to maintain a data structure describing the active portion of the execution tree. Requests to initiate a new task are deferred if the activity level is sufficiently high, and recorded at the appropriate place in the representation of the active subtree maintained by the resource management system. When the activity level falls below some threshold, the lowest, leftmost pending request is granted. Loops in this machine are driven by "proliferate" operators, which produce all n values, say, of the iteration variable. These are made to produce values in small chunks and must go through a task request in between. This approximates the effect of k-bounded loops to some extent, but the resource management system is still making a huge number of scheduling decisions on fairly small items.

Although experimental data from a small set of programs looks encouraging, it is based on programs in a language setting that is considerably more restrictive than that in *Id*, which we are addressing. In particular, functions and data structures are strict in SISAL[69]. Ruggiero's approach avoids the problem of losing the ordering tasks within the tree, but at a cost of considerable complexity. In theory, it is still subject to the hazards of over-eager initiation and over-reaction in periods of low parallelism. Moreover, unlike parental suspension, deferring a request does not prevent the requester from making additional requests. We might consider a loop which invokes a task that consumes and produces successive versions of a non-strict data

structure in each iteration, such as a relaxation algorithm. All the iterations of the outer loop will unfold, presenting a huge number of task requests and data structure allocation requests. Only those pertaining to the early iterations are meaningful. We will see in Chapter 8 that, in fact, the request stream can be very noisy; a large fraction of the tasks are uninteresting and could be dealt with by static analysis.

Although we focus on a technique that pertains to a restricted class of programs and intentionally does not take into account run-time activity levels, certainly an important avenue of future research is the fusion of this with some kind of dynamic, execution tree based approach.

1.4.5 Sarkar's partitioning

Sarkar's work in the partitioning and scheduling of parallel programs[81] deals with controlling parallelism indirectly, in the process of addressing various machine related issues. His approach is essentially to unfold the program at compile time and then to fold it back up into a statically determined number of tasks. Each task is essentially sequential. This approach breaks down for a non-strict language[86]. His partitioning criterion accounts for data structures in so far as it treats them as large objects that traverse program arcs, so the size must be estimated in order to estimate communications bandwidth. It does not deal with storage requirements directly, nor with structures that reside in some form of storage. The basic thrust of his approach is to treat the entire execution of the program as static, whereas we try to perform static analysis in order to understand its dynamic behavior.

The interesting aspect of the partitioning work from the viewpoint of controlling parallelism is the attempt to split tasks where there would otherwise be a parallelism bottleneck, or constriction point. The figure of merit used is the number of potentially concurrent tasks. Unfortunately, the data presented here indicates that in many cases a large number of tasks exist concurrently, while only a few are truly active.

1.4.6 Our approach

Our work falls somewhere in between the various efforts discussed above. We perform a substantial amount of static analysis in order to understand the dynamic behavior of the program as a whole. The results of that analysis are used to integrate resource management code into the program itself, so that execution is controlled in a quasi-dynamic fashion, although we do

not take run-time machine status indicators into account. In effect, the kinds of policies we develop start with all loops tightly constrained and selectively relax these constraints so as to get near-maximal parallelism, ignoring communication costs, within hard resource constraints. The quality of the analysis affects performance, but not correctness. We may err by overestimating the resource requirement, in which case some resources are squandered, or by underestimating a dependence, in which case resources are invested where they are poorly utilized.

The domain of programs we study is restricted to what traditionally encompasses scientific applications, although the language paradigm is very powerful. The main instrument of measurement is an idealized execution model. The "knobs" we have to turn in controlling program unfolding are the number of concurrent iterations of each loop. Our goal is to develop an understanding of the effects of these knobs and to capture this understanding in a coherent loop bounding policy.

Chapter 2

Characterizing the Inherent Behavior of Dataflow Programs

In this chapter, we present an idealized dataflow execution model, under which we later examine the behavior of various k-bounding policies. Parallelism profiles and resource profiles are introduced as a means of characterizing the inherent behavior of programs. Through several examples, we argue that a broad class of programs exhibit ample parallelism when expressed in this framework, because parallelism is exploited at all levels. However, the resource requirements of these program can be exorbitant. We define a more realistic finite-parallelism model, and observe that resource requirements remain large when little of the potential parallelism is exploited.

2.1 An Ideal Execution Model

The U-interpreter[13] provides a formal model for dynamic (tagged-token) dataflow computation. In that model, data values are carried on tokens, which also carry a *tag*. The tag specifies the particular node in the dataflow graph that is to be executed, and uniquely identifies an instance of that node. A node is permitted to execute, or *fire*, when tokens with matching tags are available on each of its input arcs, and upon firing a result token is produced for each output arc. The behavior of a node is determined entirely by the input values, the tag, the operation associated with the node, and the local connectivity of the node. Thus, a program graph defines a partial order on scheduling of instructions for a given input. High-level constructs in functional languages are captured by a specific set of graph schemas, with a simple

rule of composition. It has been shown that the programs represented by these graphs are deterministic, *i.e.*, all legal execution schedules produce the same results, and that the graphs are self-cleaning, *i.e.*, no tokens are left in the graph upon completion[14]. Furthermore, this model is maximally parallel, in a technical sense[15].

To characterize the inherent behavior of programs, we consider a particular, well-defined execution schedule, corresponding to fully eager execution. This is specified by the following:

1. *All operators take unit time.* Tokens produced by an operation appear on the output arc one time unit after the node fires.

2. *Communication is instantaneous.* Tokens produced by an operation are available at the successor nodes as soon as they are produced.

3. *An operator executes as soon as* all *its input data are available.* This implies that any number of operations can be performed in a single step, any number of data values can be transmitted between steps, and any number of tokens may be waiting for partners.

4. *The value produced by an I-structure fetch (I-fetch) operation is available one time unit after the later of the I-fetch and the I-store for the particular element accessed.*

This definition is summarized in Figure 2.1. The results of an operation are produced one time unit after the latest of the input tokens becomes available and become available at destination operations immediately after they are produced.

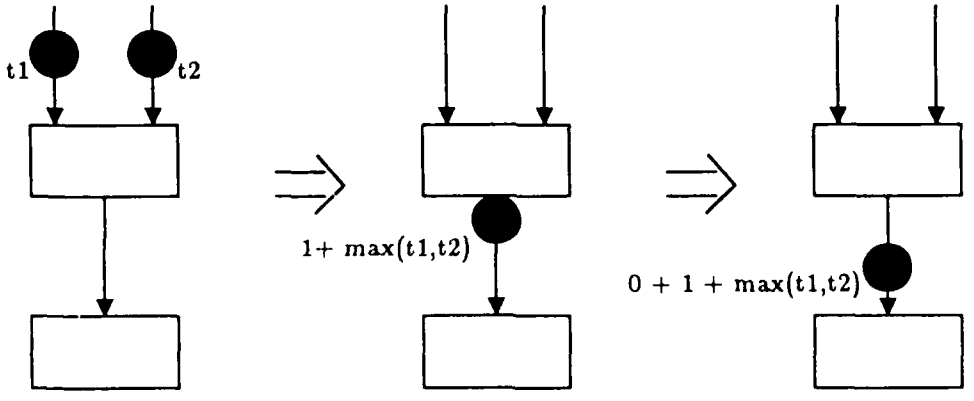


Figure 2.1: Time Behavior of the Basic Firing Rule in the Ideal Model

The *parallelism profile* for a given program on a given input is a function, $pp(t)$, which gives the number of operations fired at each step t under this ideal execution model[10]. The

execution time, denoted τ_∞, is the step beyond which $pp(t)$ is uniformly zero. Because all operations execute at the earliest possible time, this is the length of the longest chain of data-dependencies in a program, *i.e.*, the length of the *critical path*. The area under the curve $pp(t)$ is simply the total number of operations executed, denoted by τ_1 to indicate the number of steps required with parallelism of one. The ratio of these is the *average parallelism* in the program, $P_{ave} \triangleq \tau_1/\tau_\infty$.

To illustrate these definitions a simple example is given in Figure 2.2. The right-hand portion shows an arithmetic expression in *Id* and its corresponding dataflow graph. The upper-left portion shows the parallelism profile for this graph, labeled to indicate when particular nodes fire.

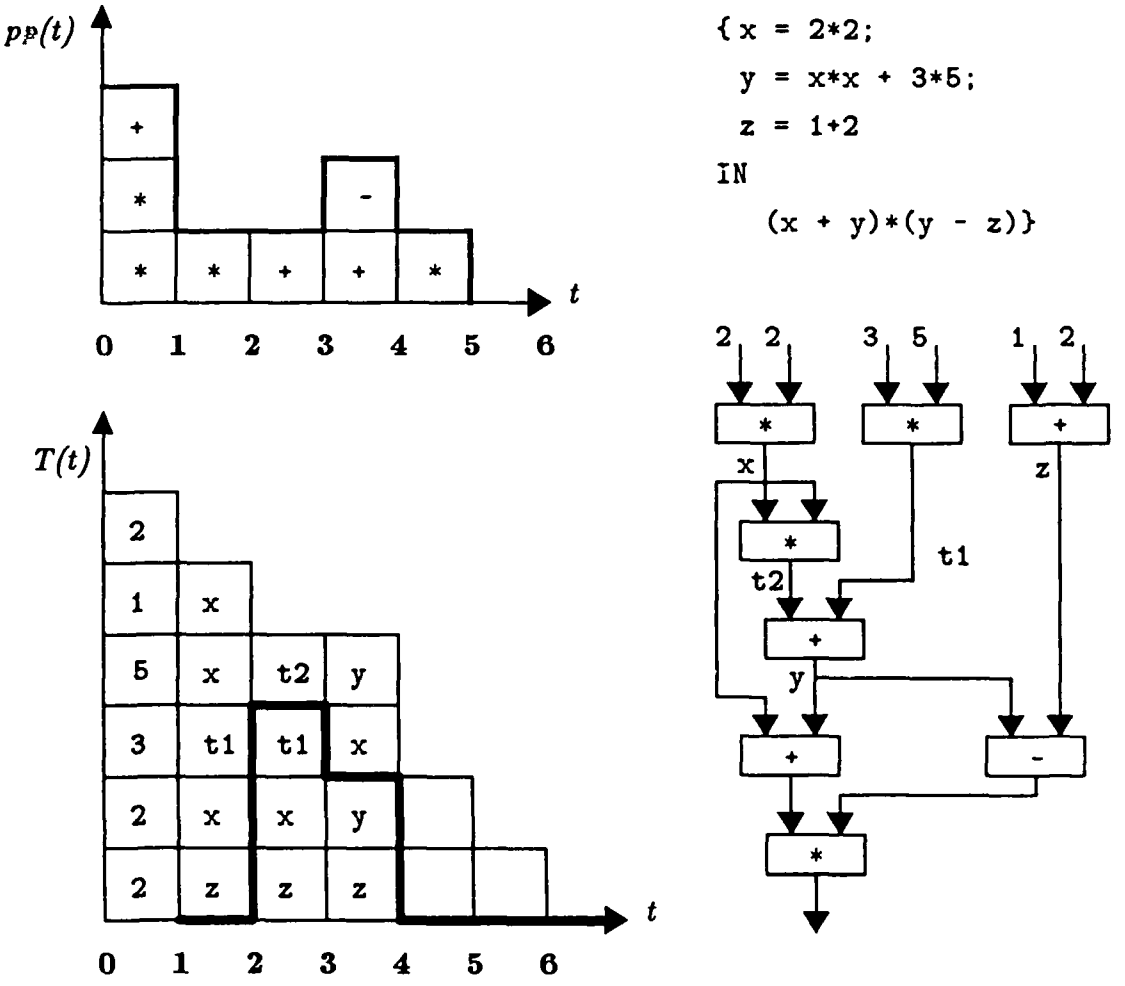


Figure 2.2: *Id* Expression, Dataflow Graph, and Associated Parallelism and Token Profiles.

It is also possible to extract a *resource profile* for various resources under ideal execution.

Each token carries a data value, and thus implies a claim on storage. The number of tokens in existence at the start of each step gives a *token-storage profile*, characterizing this resource requirement. This is shown in the lower-left portion of Figure 2.2. If we assume that tokens reside in some matching-store until their partner(s) become available, it make sense to construct a *waiting token profile*, counting only those tokens in existence which await a partner; this profile is indicated by the bold line in the token profile. We will be interested in a variety of resource profiles derived from ideal execution of real dataflow programs.

It should be noted that in making this execution model precise, including its time behavior, we have effectively made it synchronous. In a single step, all enabled activities fire, and in the next step all activities enabled by the data values produced in that step will fire. This is somewhat at odds with the practical motivations for dataflow graphs as a parallel machine language, namely, that the execution order is able to adapt dynamically to asynchronous events, such as varying execution times for primitive operations, unpredictable delays due to network contention, and the like[17]. In the ideal model, we have eliminated all sources of asynchrony by assuming unbounded processing power, communication bandwidth, and storage capacity with unit-time operations and zero latency. We will not, however, take advantage of this synchronous interpretation in any way; the graphs considered are suited for general, asynchronous execution.

To provide a better feeling for the dynamic nature of dataflow execution and gain some insight toward loop unfolding, we consider one more small example, a simple loop that computes the inner product of a row and a column of two matrices. The *Id* code for this is shown below. The dataflow graph for this loop is shown in Figure 2.3. The loop schema has a bank of *L-switch* nodes, one for each value circulated from one iteration to the next, controlled by a common predicate. A wave of tokens at the input to the *L-switches* is steered into the loop body, if the predicate evaluates to TRUE, and eventually produces a new wave for the next iteration. As soon as the predicate is evaluated, the tokens for the next wave can begin entering the loop body, even though previous iterations may still be active. The tag includes an iteration number to distinguish tokens belonging to different iterations, which the *L-switch* increments on tokens sent into the body.

```
DEF IP A B s i j = {sum = 0
                    IN {FOR k <- 1 TO s DO
                          NEXT sum = sum + A[i,k]*B[k,j];
                        FINALLY sum}};
```

The IP loop has two circulating values, k and sum. Each is a linear recurrence, with sum also dependent on k. Note, several instructions reference *loop constants*, indicated by labeled $\top$ inputs; these are variables defined outside and referenced within the loop. In this example, the descriptors for the two matrices A and B, and the scalars i, j, and s are loop constants. They are associated with the loop invocation as a whole via explicit *constant-store* operations in the prologue of the loop (not shown). Matrices are stored in row-major order, so part of the index calculation for A[i,k] has been lifted out of the loop, forming another loop constant. The $\bowtie$ node should be treated as an identity operation for the moment; its role will be explained later. The merge nodes at the top of the loop, denoted by $\otimes$, are not actually operations, but rather indicate that two arcs are connected to the same input port.

The lower portion of Figure 2.3 shows the parallelism and token storage profiles for this graph, assuming $s = 3$ and the predicate (node 1) first fires at time 0. Node numbers are shown in the profiles to aid in following the explanation. Node 2, the $\bowtie$, fires at time 1. The two *L-switches* fire at time 2, producing tokens with iteration number 1. At time 3, the increment fires (node 5), producing the next value of k, and the first parts of the two index calculations fire (nodes 6 and 7). At time 4, the predicate fires again, the *I-fetch* against A fires, along with the second part of the other index calculation. From this point on, the second iteration overlaps with the remainder of the first. Operations belonging to the even iterations are outlined in bold lines to highlight the overlap. At time 6, the *L-switch* for k fires, producing a token with iteration number 2. The other *L-switch* does not fire until time 8, at which time the predicate fires for the third time. The schedule repeats every 4 time units, until the predicate turns FALSE and the last wave drains out of the loop. Here we can see the automatic loop unfolding under data driven evaluation. The k loop gets one iteration ahead of the sum loop, while a wave of tokens moves through the fetch-and-multiply portion of the graph. After this point, summations are formed at the rate index values are generated, and we see a steady resource load of six tokens. In more realistic scenarios, *e.g.*, if the floating-point operations took longer, if latency were experienced in fetching the elements of the matrices, or if the *I-fetches* were deferred (discussed below) because the data was not available, a greater amount of unfolding would occur automatically.

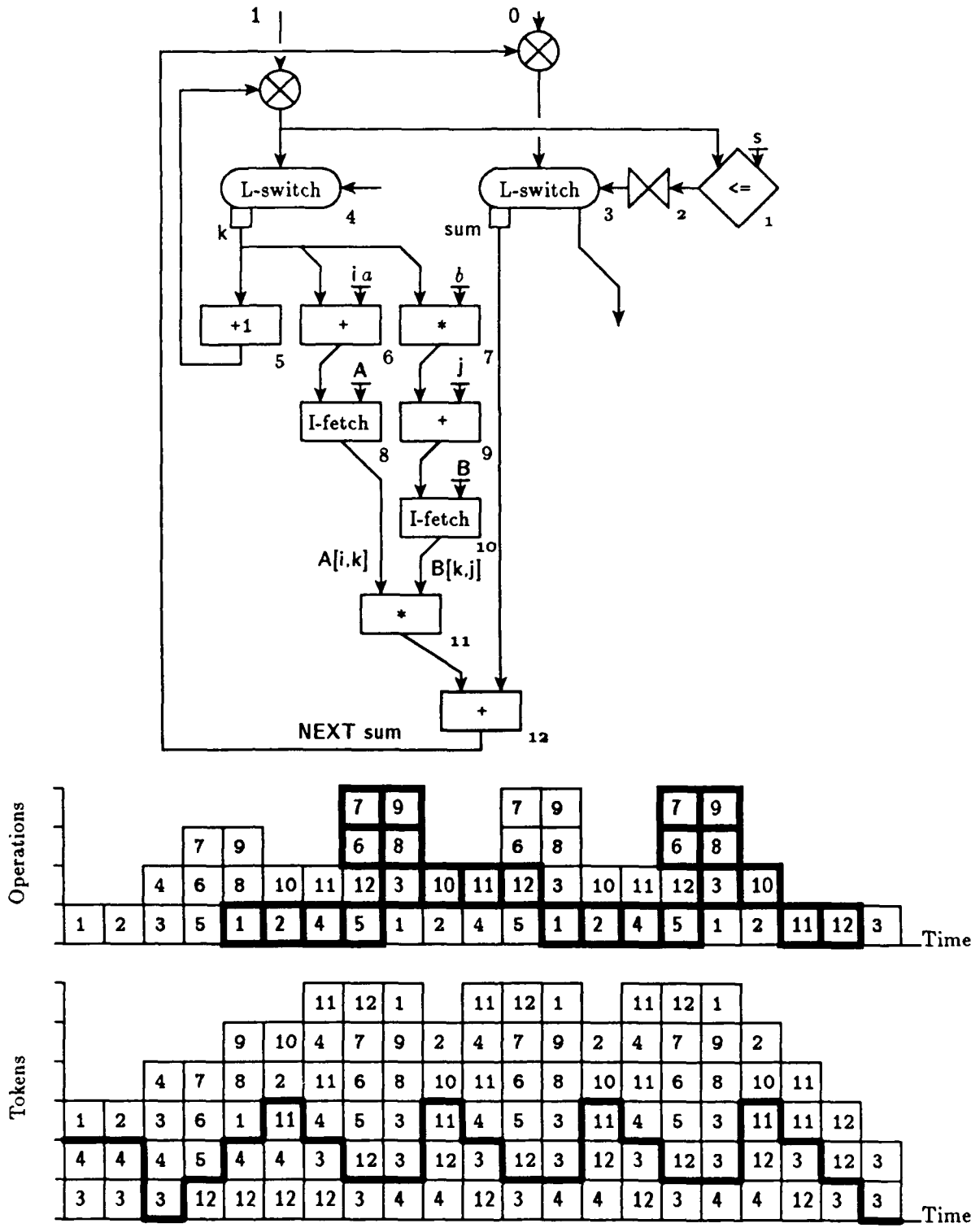


Figure 2.3: Loop Graph for IP and Associated Profiles

2.2 Parallelism and Resource Profiles for *Id* Programs

We turn now from simple examples with hand-constructed profiles to larger programs compiled from a high-level language and profiles constructed by executing the program on a graph interpreter[18]. We briefly outline the basic graph schemas and essential resources.

2.2.1 Graph generation

The dataflow graphs we consider are compiler generated from the high-level language *Id* [71, 85]. We explain the salient aspects of these graphs, along with certain subtleties, in order to place the empirical data presented later on a sound footing. As such, we must touch on a variety of low-level technicalities. The higher-level points to be gleaned are these. Given a dataflow graph and a given input, the profiles we construct are precise; they are not obtained through analysis or approximation. There is, however, a certain bias introduced in the generation of the graphs themselves, due to the particular schemas and compilation techniques employed. We desire the textual representation of a program to be as precise a description of its dataflow graph as possible. For example, the generation of an arithmetic sequence of values by iteration and by binary recursion are expressed differently. Transformations from one to the other should be viewed as a source-to-source transformation at some higher level, as should optimizations which involve algebraic identities, *e.g.*, strength reduction. Most importantly, no arbitrary constraints are placed on the kinds of parallelism that can be expressed and exploited. The language is block structured, where all blocks are *non-strict*, so the body of a user-defined function, loop, conditional, or arithmetic expression can begin execution as soon as *any* of its inputs are available. Moreover, it can execute in parallel with its parent or surrounding context. This allows parallelism to be exploited at all levels. Even data structures are non-strict, so producers and consumers can execute in parallel. There is no need for compiler analysis to uncover parallelism; it is expressed directly in the program and extracted automatically by the machine. In fact, it is generally difficult to determine a sequential ordering for an *Id* program at compile time[86]. Program and data are separate. Graphs are re-entrant.

Only unary and binary operations are permitted, in order to limit the complexity of the matching operation. No hard limit is placed on the number of output arcs of a node, *i.e.*, the number of copies of the result to be generated, however, operators that produce a dynami-

cally determined number of results, such as the Manchester machine's *proliferate*[44], are not permitted. The fan-out of a node is basically the number of right-hand side occurrences of the corresponding variable in the text of the program. In order to detect termination of an invocation, all nodes have at least one output, although the output may be merely a signal indicating that the node has fired.

Each of the schemas discussed below are *well-behaved*, in the sense that a single wave of tokens on the input arcs produces a single wave of tokens on the output arcs, and when a token emerges on every output arc, no tokens remain in the graph. The graphs generated by legal composition of these schemas are also well-behaved. Thus, we do not consider arbitrary graphs, but only a restricted class of graphs generated from the high-level language *Id*.

Arithmetic and logical expressions are represented by the natural directed acyclic graph, as illustrated in Figure 2.2. Conditional expressions employ a *switch* operation, which steers its data input to one or another set of output arcs, based on a control input. Each value used in either arm of a conditional is steered through a switch, controlled by the predicate value. Once the predicate is determined, values enter the appropriate arm of the conditional in whatever order they become available. The arms can be arbitrary well-behaved graphs. Unused outputs within the conditional are reduced in a tree to produce an additional signal output[26, 85].

Each user-defined function or nested loop in the program is represented as a separate graph, called a *code-block*. Functions in *Id* are non-strict and can begin execution before any arguments are available. Invocation of a code-block involves acquiring a *context*, or unique identifier, which is carried by all tokens belonging to the particular invocation. Arguments are delivered to the new invocation by a *send* operation, which places the context identifier for the invocation in the tag part of its input token. Activity is triggered within the new context as each of the operands arrive. The computation within a function can proceed in parallel with computation of the arguments to the function. Results are similarly delivered to the parent context via *send* operations.

The basic structure of the loop schema is illustrated by Figure 2.3, above. The loop body and predicate can be an arbitrary well-behaved graphs, where the predicate produces a single boolean result. A bank of *L-switches* steers waves of values into the body, as determined by the predicate, and increments the iteration numbers on tokens sent to the body to distinguish them from logically previous iterations. Each of the *L-switches* fires independently, so the loop values

need not circulate in well-defined waves. Some loop values may get many iterations ahead of the others, as we observed in constructing the parallelism profile for the inner product loop.

Values which are used within a loop, but constant over all iteration of the loop are not circulated, but placed in a special loop-constant area before the first iteration is allowed to begin. Thus, loops are strict in the loop-constant arguments. The language treats all iterations as strict, so an iteration may depend only on logically previous ones. This subtlety is key for k-bounding. For the purposes of constructing resource profiles, we say an iteration starts when the $\bowtie$ fires, delivering the predicate value for the iteration to the bank of *L-switches*, and ends when no tokens remain for the iteration. The basic graph schemas are structured so that it is possible to detect completion in this sense.

2.2.2 Resources

The resources associated with an executing dataflow program differ somewhat from more conventional settings. Token storage is the most apparent resource, because it is manifest in the graph; the other resources require more explanation. As mentioned above, associated with each invocation of a code-block is a context. The context is allocated by the parent, prior to sending arguments, and deallocated when all results, including the termination signal, are returned[85]. Thus, contexts correspond roughly to stack frames in a conventional language implementation. The primary difference is that the collection of extant contexts form a tree[26], rather than a stack, and the entire tree is active, not just the leaves. A context is a resource in two respects. First, it denotes a region of the tag space used for synchronizing operations within an invocation. Secondly, storage may be associated with the context directly. In the MIT Tagged-Token Dataflow Architecture, each context has an associated code-block register, containing a pointer to the graph and to a constant data area[9].

As remarked above, an invocation of a loop code-block may involve many concurrent iterations, and these are distinguished by an iteration field in the tag. Although allocation and deallocation is not explicit for iterations, as it is for contexts, each iteration does place an implicit claim on certain resources —it implies use of a certain region of the tag space and at any time a number of tokens are extant for each active iteration. We will see in a later chapter that k-bounded loops allow this iteration field to be kept to a fixed size. Once the tag space is used in a fairly dense manner, the next logical architectural step is to equate a tag with an address.

This idea has been thoroughly developed by Papadopoulos in his Explicit Token-Store (ETS) dataflow architecture[74]. In the ETS approach, each active acyclic code-block invocation and each concurrent iteration has an associated *activation frame*, providing local storage for tokens and constants. Since frames subsume the role of contexts, the two forms of local storage we consider are waiting tokens and activation frames.

Finally, arrays in *Id* are represented as I-structures[19], *i.e.*, sequences of write-once slots. An I-structure is explicitly allocated and filled. It should be viewed as residing in storage, not as a conglomerate of values on an arc. It is a non-strict data structure, as it can be used before it is completely defined. Initially, each slot is empty, to be filled by an *I-store* operation. If a fetch for a slot occurs before the corresponding store, the fetch is deferred until the store takes place. This gives rise to the subtlety addressed by the last item in our description of the ideal execution model — the result of a fetch is available one time unit after the later of the *I-fetch* and the corresponding *I-store*. For the purposes of constructing I-structure storage profiles, a structure is in use from the time it is allocated until the time it is deallocated. Since there is no automatic storage reclamation in the graph interpreter used to construct execution profiles, I-structure profiles are primarily suggestive of the pattern of allocation.

2.3 Examples

In this section we illustrate the level of potential parallelism present when parallelism is exposed at all levels through several example programs. In the process, we build an understanding of the parallelism/resource problem.

2.3.1 Matrix-multiply

The first *Id* program we consider is a simple matrix multiplication example (shown below) using the standard triply nested loop formulation, where `C[i,j]` is the inner product of row `i` of `A` and column `j` of `B` computed by `IP` (discussed above).

This example is small enough that one can develop reasonable intuition and check that the results are meaningful, and yet it intimates the power of our method of characterizing program behavior.

```
DEF MatrixMult A B n s m =
  {c = I_matrix((1,n),(1,m));
   {FOR i <- 1 to n do
      {FOR j <- 1 to m do
         c[i,j] = IP A B s i j}}
   in c};
```

Figure 2.4 shows the ideal profiles for this program on two 16×16 matrices. In the upper portion is the parallelism profile, expressed as the number of concurrent operations per step. Also shown is the number of floating-point operations. This program has $\tau_1 = 75,819$ and $\tau_\infty = 264$. Thus, $P_{ave} = 287$, about 1/3 of the peak parallelism ($P_{max} = 904$). We can see that parallel activity is not generated instantaneously; it takes a while for the full parallel activity to build up and also for this activity to taper off. The algorithm specifies n^2 independent inner products, each involving a linear recurrence of $O(n)$. This correlates with the profile, and we may conclude that parallelism is effectively exploited in both outer loop levels. When the problem is scaled to 32×32 matrices the profile maintains the same general shape, reaching $P_{max} = 3,404$ with $\tau_\infty = 456$ (roughly double) and $P_{ave} = 1,127$. In fact, we will argue in Chapter 5 that τ_∞ for this program tends toward $12n$, as n increases.

The lower portion of Figure 2.4 shows three resource profiles for this program: waiting tokens, activation frames, and I-structure storage. The last of these is simply the size of the result matrix. The other two are roughly proportional to the amount of exposed parallelism, with two tokens and one frame per operation ($A_{max} = 915, W_{max} = 2,187$). This is not surprising, since each active operation requires storage for its arguments, but one might ask how this changes when the machine can only exploit a fraction of the exposed parallelism. We will see below that under fair scheduling the resource requirements remain essentially unchanged in this situation.

The parallelism profile shown in Figure 2.4 describes the inherent behavior of this particular formulation; it is not necessarily the maximum parallelism that can be achieved in multiplying matrices. If we intended to compute the inner product as a binary recursive tree, we could do so, but it would be expressed by a different formulation, as shown by `rip` below. The profiles for Matrix Multiply using this version of the inner product on 16×16 matrices are shown in Figure 2.5. The critical path is reduced from 264 to 238, and the total instruction count is essentially the same, giving $P_{ave} = 311$. The resource requirement is larger in the recursive version, ($A_{max} = 961, W_{max} = 3,420$). For 32×32 matrices, $\tau_\infty = 423$ and $P_{ave} = 1,225$, so

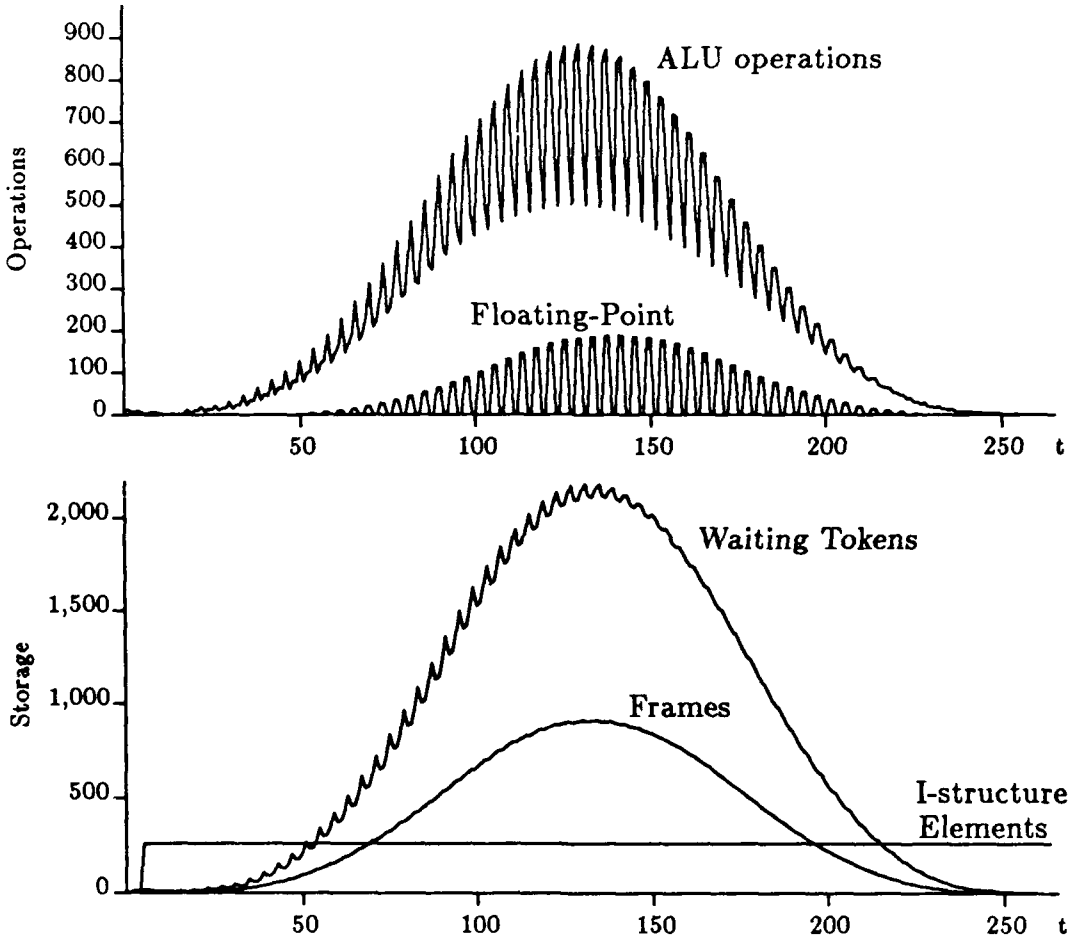


Figure 2.4: Ideal Parallelism and Resource Profiles for MatrixMult on 16×16 matrices

the difference in the behavior of the two programs are still fairly small. In many large scientific applications, we find that ample parallelism is available with traditional iterative formulations.

```
DEF rip A B i j low high =
  if low >= high then (A[i,low]*B[low,j])
  else mid = floor ((low + high) / 2);
       in (rip A B i j low mid) + (rip A B i j (mid+1) high);
```

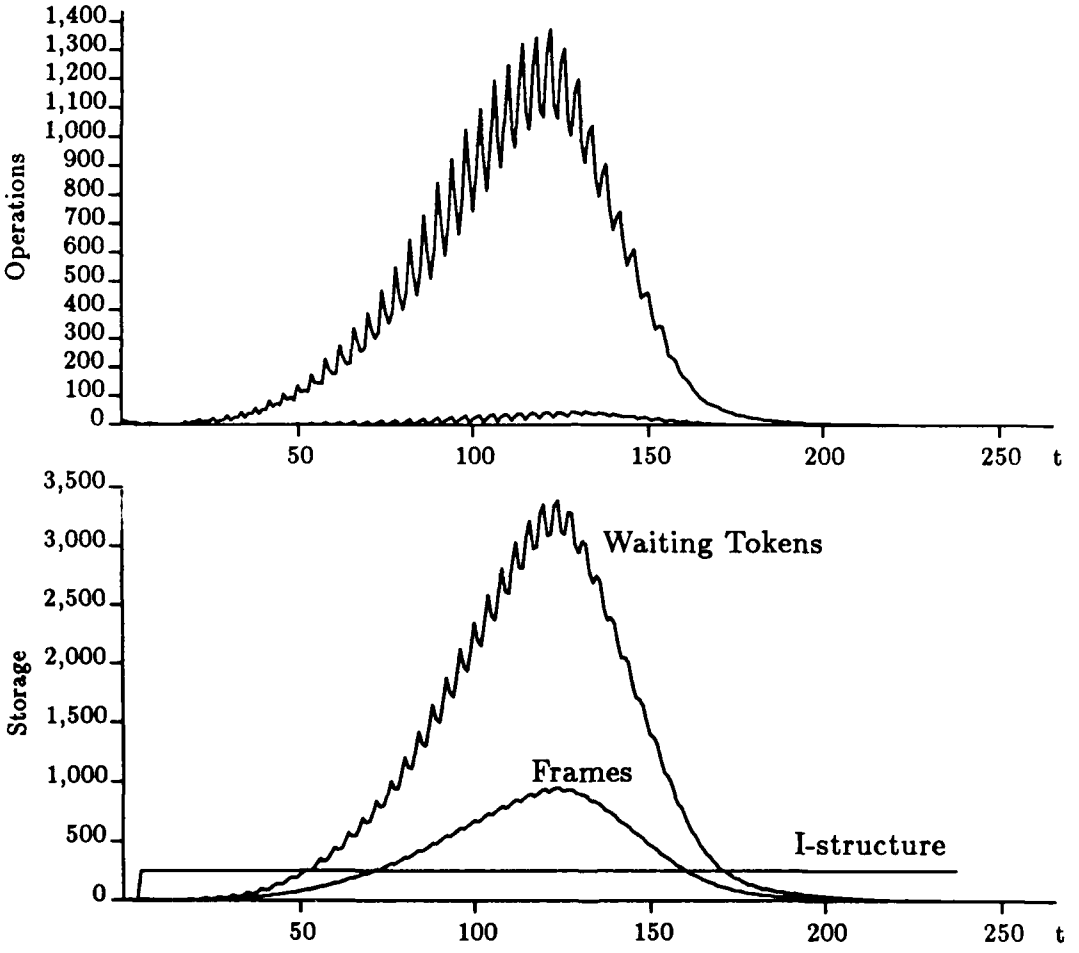


Figure 2.5: Ideal Parallelism and Resource Profiles for MatrixMult with Recursive Inner Product on 16 × 16 Matrices

2.3.2 Relaxation

As a somewhat more interesting example, we consider 2D relaxation using a 9-point stencil, as might be used to model heat flow in a rectangular region with insulators at the boundaries. The *Id* code is shown below and the parallelism and resource profiles are shown in Figure 2.6

for a 16×16 mesh. The 9-point stencil is computed by the first function.[1] The interior portion is computed by a doubly nested loop (**interior**), with single loops (**sides** and **top_bottom**) computing the edges. For this example we have, $P_{max} = 188$, $\tau_\infty = 195$ and $\tau_1 = 12,811$, giving $P_{ave} = 65$. On the resource side, $A_{max} = 130$ and $W_{max} = 299$. This example demonstrates the virtue of a programming methodology that encourages creating new objects, rather than updating existing ones. If this algorithm were written to operate in-place in a conventional language, the programmer would have to introduce temporaries to hold the old values of the points represented by the top and left edges of the stencil. This becomes awkward to write and obscures the potential parallelism by introducing anti-dependences. Removing these anti-dependences at compile time in order to uncover the potential parallelism is quite difficult. The more likely scenario is that the programmer allows the stencil to pick up some old points and some new points; this changes the mathematics from a Jacobi method to a Gauss-Seidel method[31]. The latter does have better convergence properties, but admits less parallelism. In *Id* either can be expressed with equal ease and the differences in the two algorithms are explicit in the program formulation.

```
%%% 2D Relaxation using Nine-point Backward Difference Operator
%%% with reflective boundary conditions.
defsubst stencil A il i ih jl j jh =
   A[i,j] - ((  A[il,jl] +  4*A[il,j] +   A[il,jh]) +
             (4*A[i ,jl] - 20*A[i ,j] + 4*A[i ,jh]) +
             (  A[ih,jl] +  4*A[ih,j] +   A[ih,jh])) * 0.05;

defsubst sides R A l1 u1 l2 u2  =
  {for i <- l1+1 to u1-1  do
     R[i,l2] = stencil A (i+1) i (i+1) (l2+1) l2 (l2+1);       % first col
     R[i,u2] = stencil A (i+1) i (i+1) (u2-1) u2 (u2-1)};      % last col

defsubst top_bottom R A l1 u1 l2 u2  =
  {for j <- l2+1 to u2-1 do
     R[l1,j] = stencil A (l1+1) l1 (l1+1) (j-1) j (j+1);       % first row
     R[u1,j] = stencil A (u1-1) u1 (u1-1) (j-1) j (j+1)};      % last row

defsubst interior R A l1 u1 l2 u2  =
   {for i <- l1+1 to u1-1 do                                   % interior
      {for j <- l2+1 to u2-1 do
         R[i,j] = stencil A (i-1) i (i+1) (j-1) j (j+1)}};
```

[1] The **defsubst** keyword indicates that it is permissable to expand the function graph "*in* line". For the functions containing loops, it merely enables optimization of the call boundary.

```
defsubst relax_step A  =
  {((l1,u1),(l2,u2)) = (2D_bounds A);
   R = I_Matrix ((l1,u1),(l2,u2));
   R[l1,l2] = stencil A (l1+1) l1 (l1+1) (l2+1) l2 (l2+1);      % Corners
   R[l1,u2] = stencil A (l1+1) l1 (l1+1) (u2-1) u2 (u2-1);
   R[u1,l2] = stencil A (u1-1) u1 (u1-1) (l2+1) l2 (l2+1);
   R[u1,u2] = stencil A (u1-1) u1 (u1-1) (u2-1) u2 (u2-1);
   call sides      R A l1 u1 l2 u2;
   call top_bottom R A l1 u1 l2 u2;
   call interior   R A l1 u1 l2 u2
  in R};
```

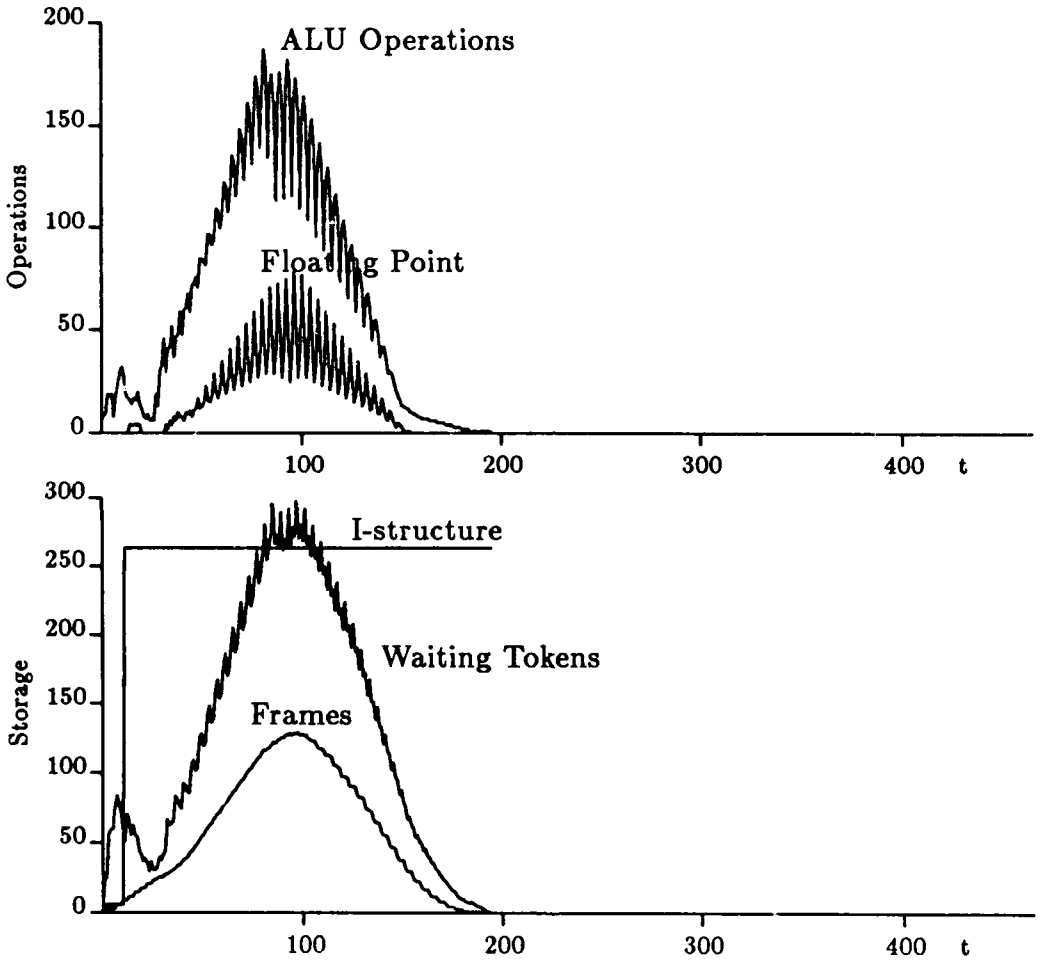


Figure 2.6: Parallelism and Resource Profiles for Relax_step on a 16 × 16 Mesh

Even though all the loops are essentially FORALL constructs, the recurrence involved in generating the index values has a significant effect. This can be mediated somewhat by static unrolling of the inner loops. Observe, however, that the loops handling the boundary conditions and the nest dealing with the interior completely overlap.

It is more interesting to examine the behavior of several iterations of `relax_step`, as represented by the successive over-relaxation program shown in *Id* below.

```
def relax A steps =
  {for i <- 1 to steps  do
     next A = relax_step A;
   finally A};
```

The parallelism and resource profiles for 16 iterations on a 16×16 mesh is shown in Figure 2.7 on the same time-scale as Figure 2.6. Successive iterations overlap, so the critical path only increases by a factor of 2.4 ($\tau_\infty = 465$). The average parallelism is now 920. This demonstrates the power of I-structures, which provide element-by-element synchronization to allow concurrent production and consumption. On the other hand, the local storage requirement has increased 11-fold ($A_{max} = 1,718$ and $W_{max} = 3,430$) and we now have 16 versions of the mesh extant concurrently.

The leveling of the I-structure profile in this example is somewhat misleading and deserves explanation. What we see from this profile is that the structures are all allocated early in the execution of the program. The profile levels at this point, because our ideal model does not assume automatic storage reclamation to deallocate inaccessible structures. When structures are deallocated as soon as they are no longer needed, the I-structure profile falls as shown in Figure 2.8.[2] Observe that the peak resource requirement is unchanged.

If we limit the number of concurrent iterations of the outermost loop to some k, at most $k + 1$ versions of the mesh will be in use at any time. Rather than allocate and reclaim these structures, it will be possible simply to circulate them within the graph. This provides a means of obtaining a clean problem formulation, which creates new structures, rather than updating old ones, while preserving the ability to reason about the space-complexity of a program.

If we embellish our relaxation example to include testing for convergence using the maximum absolute difference over the mesh, the behavior of the program changes dramatically, as shown in Figure 2.9. The opportunity for overlapping successive iterations is lost, because the predicate depends on the value at every point in the mesh of the previous iteration. To gain more parallelism it would be necessary to perform several relaxation steps between convergence tests. Note, however, that the convergence test and the relaxation step do overlap. The natural limit

[2]This profile was generated by inserting a annotation into the program text declaring the lifetime of the structure and bringing the first iteration out of the loop.

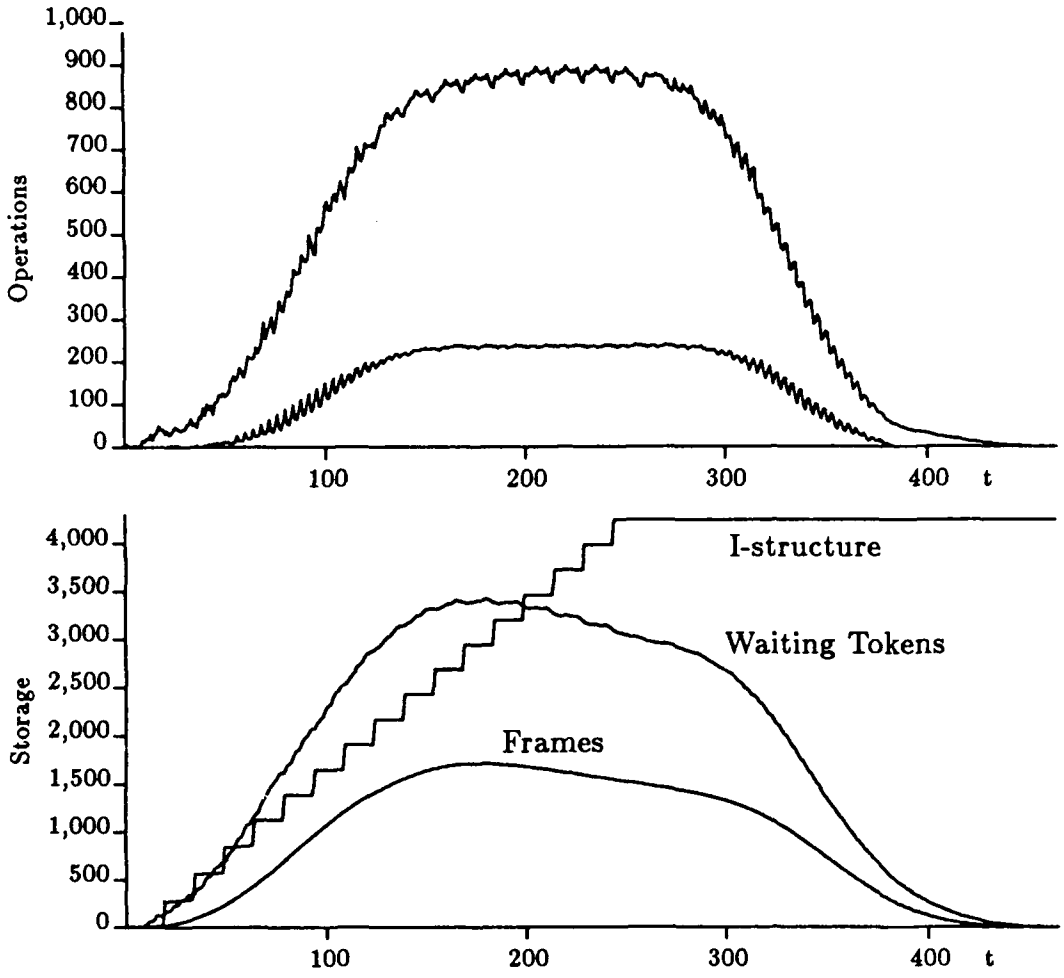


Figure 2.7: Parallelism and Resource Profiles for 16 Iterations of Relax on a 16 × 16 Mesh

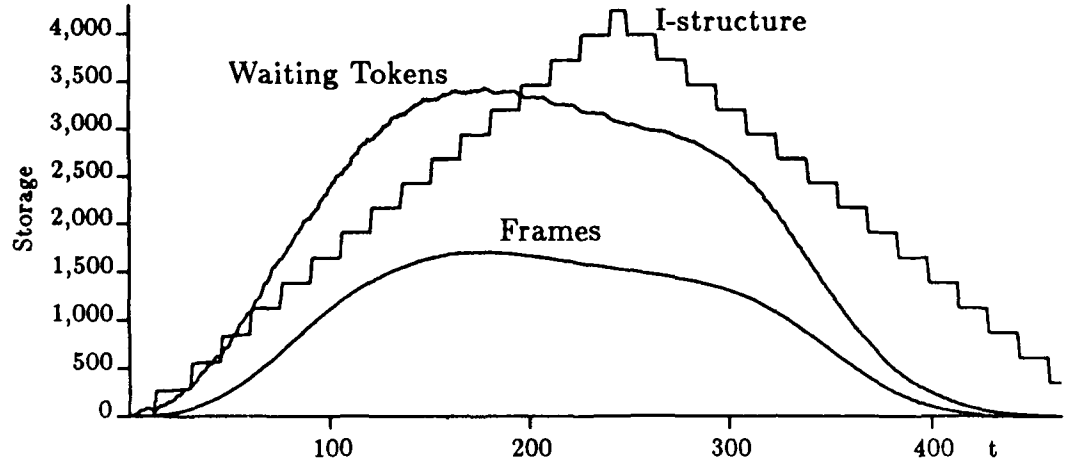


Figure 2.8: Resource Profiles for 16 Iterations of Relax on a 16 × 16 Mesh with Structure Reclamation

on the unfolding of the outer loop is similar to the effect that k-bounding has in general and likewise dictates the number of concurrent versions of the mesh that can be utilized.

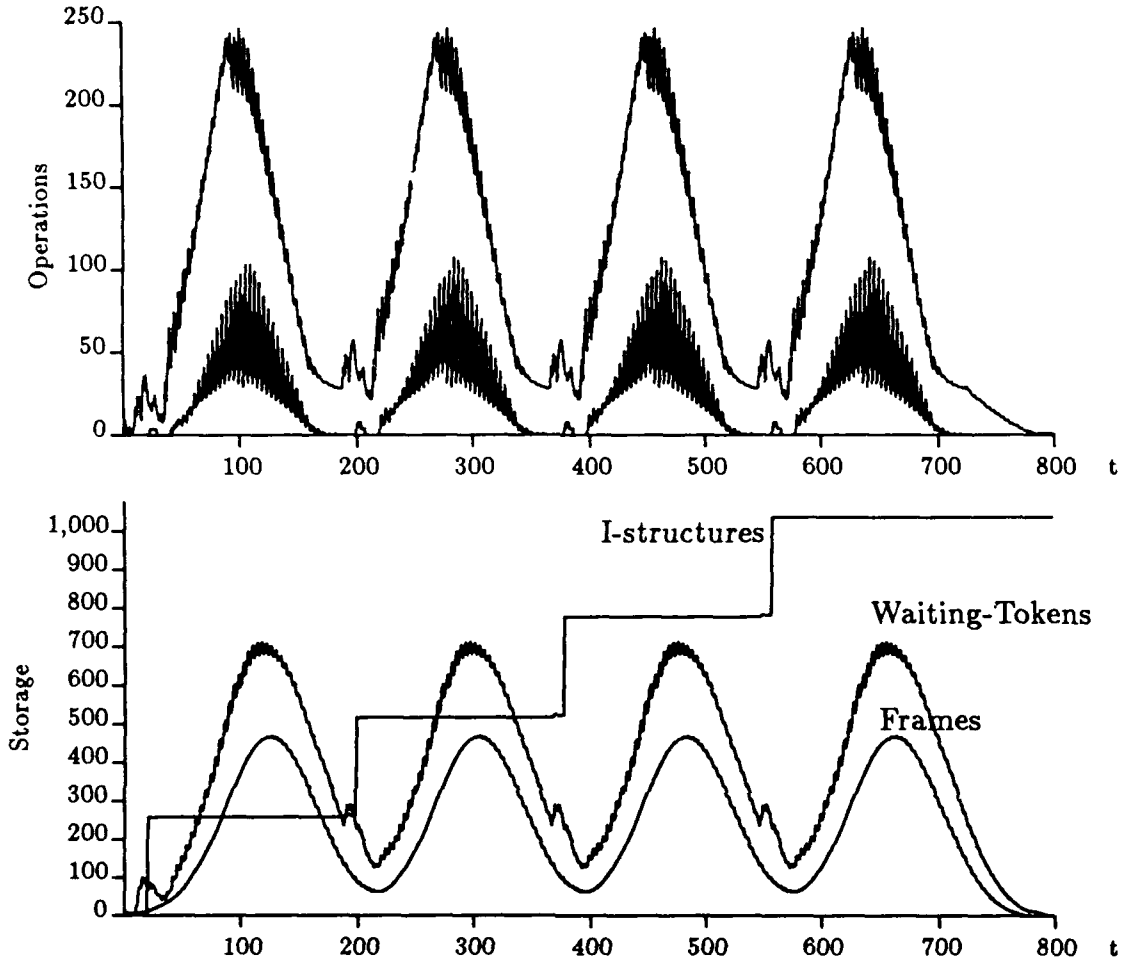


Figure 2.9: Parallelism and Resource Profiles for 4 Iterations of Relax on a 16 × 16 Mesh with a Convergence Test

2.3.3 LUP decomposition

More complex behavior is exhibited by LU decomposition with partial pivoting, due to conditionals, dynamically changing iteration counts, and overlapping phases of pivot selection and row elimination. The *Id* version of this program appears in the literature[11] and is not shown here. It was put forward as an example where write-once storage is inefficient at first glance, but proves to be much less so with a little programming care. The parallelism and resource profiles for a 20×20 problem is shown in Figure 2.10; this uses the quasi-random numbers of the

Linpack benchmark[39] as the element values. Parallelism decreases as the size of the submatrix diminishes, but the linear search for the pivot row is largely masked by the last portion of the previous row elimination. Although token frame requirements follow the general envelope of the parallelism profile, this computation is much more resource intensive than those discussed above, with nearly 20 tokens per enabled operation.

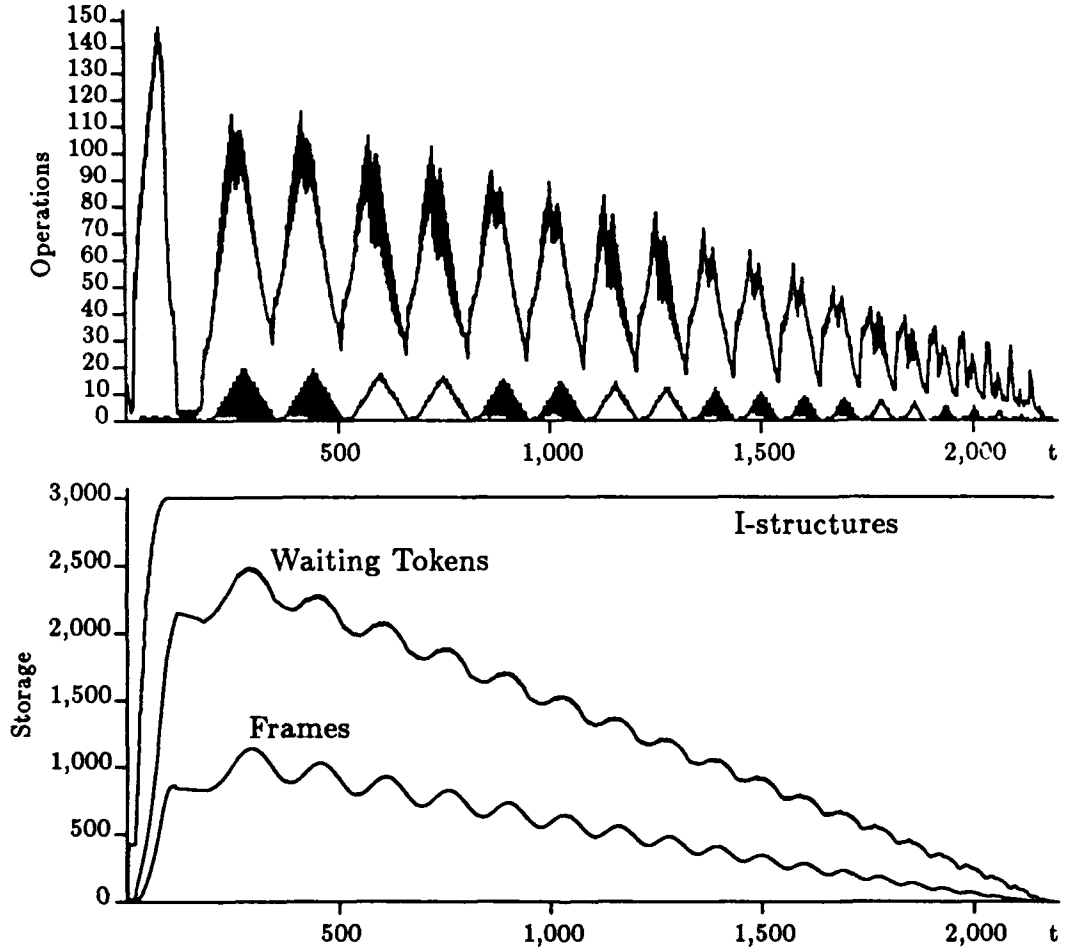


Figure 2.10: Parallelism and Resource Profiles for 20 × 20 LUP Decomposition

It is possible to write this program in a more declarative style where L, U, and P are defined in terms of one another by their mathematical relation, $A = LUP$. The interesting point here is that since the definitions are mutually recursive, the loops can only make progress by executing together. The resulting program has no valid top-to-bottom left-to-right sequential ordering. This situation arises due to non-strictness and has been studied extensively by Traub[86]. It presents serious problems when generating code for sequential machines. For our purposes, the

important fact is that, in general, we can not force one loop to execute before another.

2.3.4 Simple

As a nearly full-scale application, we examine a hydrodynamics and heat conduction simulation, known as the Simple code[25], that has been used as an application benchmark for high-performance computers. The problem is to simulate the behavior of fluid in a sphere, using a Lagrangian formulation. To simplify the problem, only a semi-circular cross-sectional area is considered. The program does not vectorize easily, due to extensive table lookups within key inner loops. The particular version we use was developed to demonstrate the possibility of programming scientific applications at a very abstract level, using higher-order functions and I-structures[11]. A declarative style is adopted in which node or zone formulae are applied to generalized co-ordinate functions over finite ranges. The resulting program directly reflects the equations describing the physics of the problem. This is quite far from the versions of this program studied elsewhere[65, 78], which employ a very low-level programming style.

Figure 2.11 shows the parallelism and resource profiles of Simple, for four iterations on a 16×16 mesh. We have $\tau_\infty = 2,969$ and $\tau_1 = 3,925,012$, giving $P_{ave} = 1,322$. The parallelism exposed here is substantial, especially considering the small problem size. Buried in this profile are some 13 distinct phases of computation, including forward and back solvers along one dimension of the mesh and then the other. The peak parallelism is 11,912, but it is clear from the profile that the amount of parallel activity varies dramatically over time.

The resource profiles, however, are cause for alarm; the resource requirement is not proportional to the exposed parallelism, as in the previous examples. It explodes when the program begins and then tapers down as the program progresses. The problem here is that the outer loop unfolds completely, allocating a version of the mesh, which includes 25 zone and nodal quantities over an $n \times n$ range, for every time-step, and spawning off the computation comprising each step. Because functions and data structures are non-strict, the computation for each step unfolds almost completely, even though that for the later steps has no data to operate on. This generates a peak resource requirement of $A_{max} = 53,777$ and $W_{max} = 260,554$! I-fetches are issued against the preceding version of the mesh, and then the computation effectively suspends until data becomes available. As data moves through the sequence of meshes, the computation for the various steps completes, finally releasing resources. This causes the

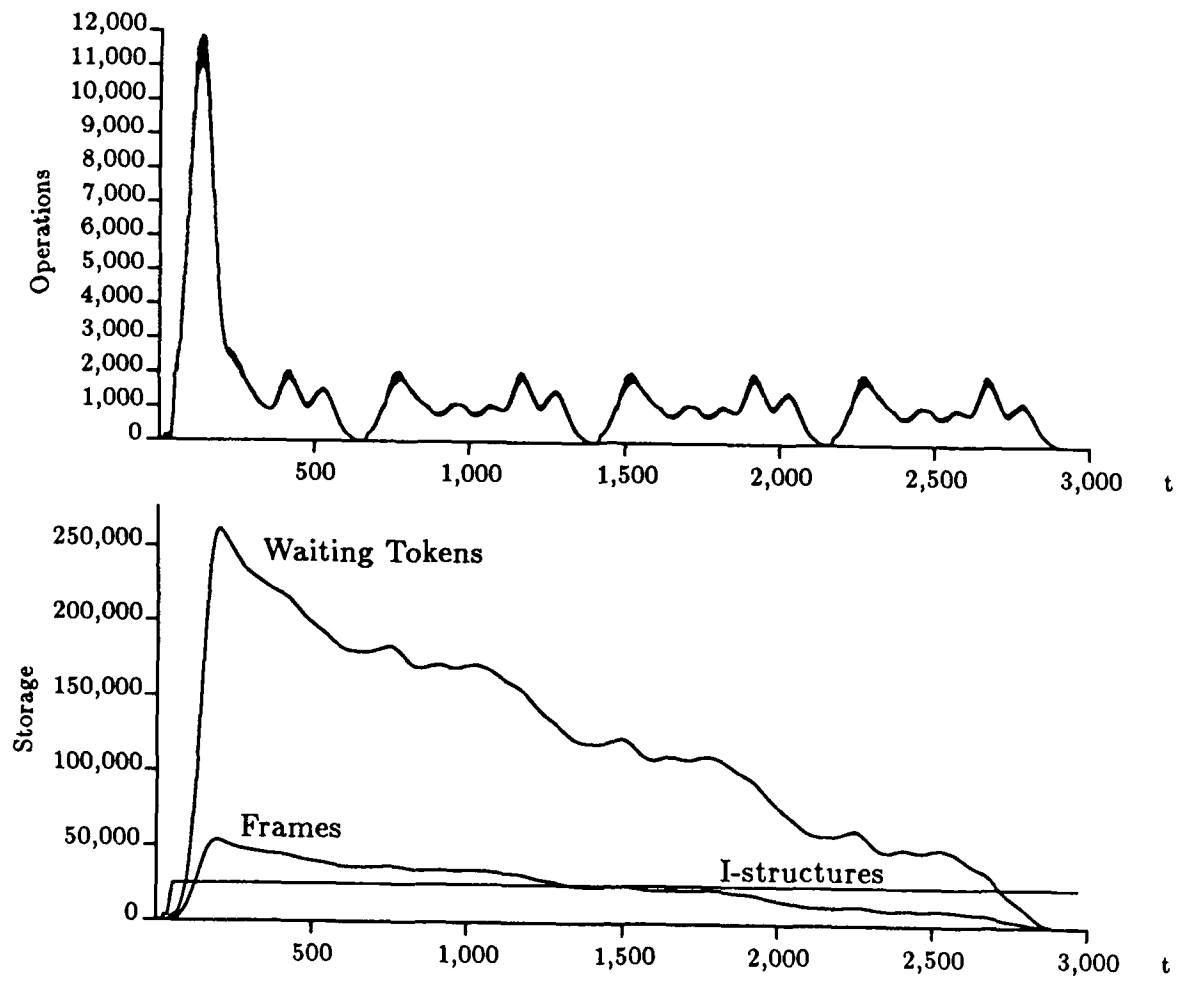


Figure 2.11: Parallelism and Resource Profiles for Simple, 16 × 16 for 4 Steps

stair-step appearance of the resource profiles. There is, in fact, a scalar value produced toward the end of a time-step, upon which much of the computation in the next iteration depends. However, neither the predicate nor the allocation of resources depend on this value, so the loops unfold in this rather alarming manner. We call the unfolding in this case *useless parallelism.*

This situation would be disastrous in a large scientific application, because the mesh is likely to be very large and many versions of it are generated. A typical problem size for Simple would be $100,000$ iterations on a 100×100 mesh. Even if I-structures were reclaimed the instant they became inaccessible, we would see the same peak I-structure storage requirement, because references to all the versions of the structure do co-exist, as we observed in the succesive relaxation example. As bad the I-structure profile appears, notice how it is exceeded by token storage requirement; both are unacceptable.

2.4 A Finite-Parallelism Execution Model

We have seen that the dynamic dataflow approach exposes tremendous parallelism, because parallelism is exposed at all levels. We have also seen that the resource requirement of such programs is at least proportional to the exposed parallelism, and in some applications it is much worse. In this section, we argue that the situation does not improve when only a fraction of the potential parallelism is exploited. Thus, we claim that the problem is not simply an artifact of our unrealistic model, but a serious pragmatic concern.

Our "finite-processor" model is specified by the following:

1. *All operators take unit time.* Tokens produced by an operation appear on the output arc one time unit after the node fires.

2. *Communication is subject to some fixed latency L.* Tokens produced by an operation at time t are available at the successor nodes at time $t + L$.

3. *An operator executes only when all its input data are available, but no more than a fixed number P execute in a single step.* When more than P operations are enabled, FIFO scheduling is assumed.

4. *The value produced by an I-structure fetch (I-fetch) operation is available $2L$ time units after the request is produced by the later of the I-fetch and the I-store for the particular*

element accessed.

This models P processors only in a very loose sense. It stipulates that the maximum parallelism that can be exploited is P, and that every value produced is subject to a latency penalty, as if all communication were external. In making the model more realistic, we have also made it less precise. Since P operations are chosen from the set of enabled operations, a large class of execution schedules are allowed for a given program on a given input. The one carried out by our graph interpreter is determined by the FIFO rule and an arbitrary ordering of output arcs for each node, as produced by the compiler.

Figure 2.12 shows the parallelism and resource profiles for Matrix Multiply (16×16) under this finite-processor model, where only a small amount of parallelism can be exploited ($P = 10, L = 2$). The critical path has increased by a factor of nearly 30 over ideal execution ($\tau_\infty = 7,737$), but the peak resource load is undiminished ($W_{max} = 2,406$ and $A_{max} = 922$). In fact, the average resource load has essentially doubled, from 826 to 1,583 for tokens and from 347 to 648 for frames.

We observe this phenomenon in most programs. This should not be surprising since execution under the finite-processor model essentially follows the schedule of the ideal model, except computational resources are multiplexed across the collection of enabled activities. Although we do not pursue the point here, the resource behavior does not change considerably when the model is made more realistic by actually mapping enabled operations onto a fixed set of processors. We call this situation *excessive parallelism*, because the program unfolds in way that exposes parallelism far beyond what the machine can exploit. To reduce the resource requirements significantly, it is necessary to exert considerable control over how the program unfolds.

2.5 Conclusions

We have shown through several examples that the dataflow graphs generated from *Id* programs allow parallelism to be exploited at all levels, *i.e.*, in nested loops, among collections of user-defined functions, within conditionals, and even in producer-consumer relationships. Furthermore, when executed in a dynamic dataflow paradigm, often a tremendous amount of parallelism can be exploited. However, in doing so, the resource requirements can be exorbitant.

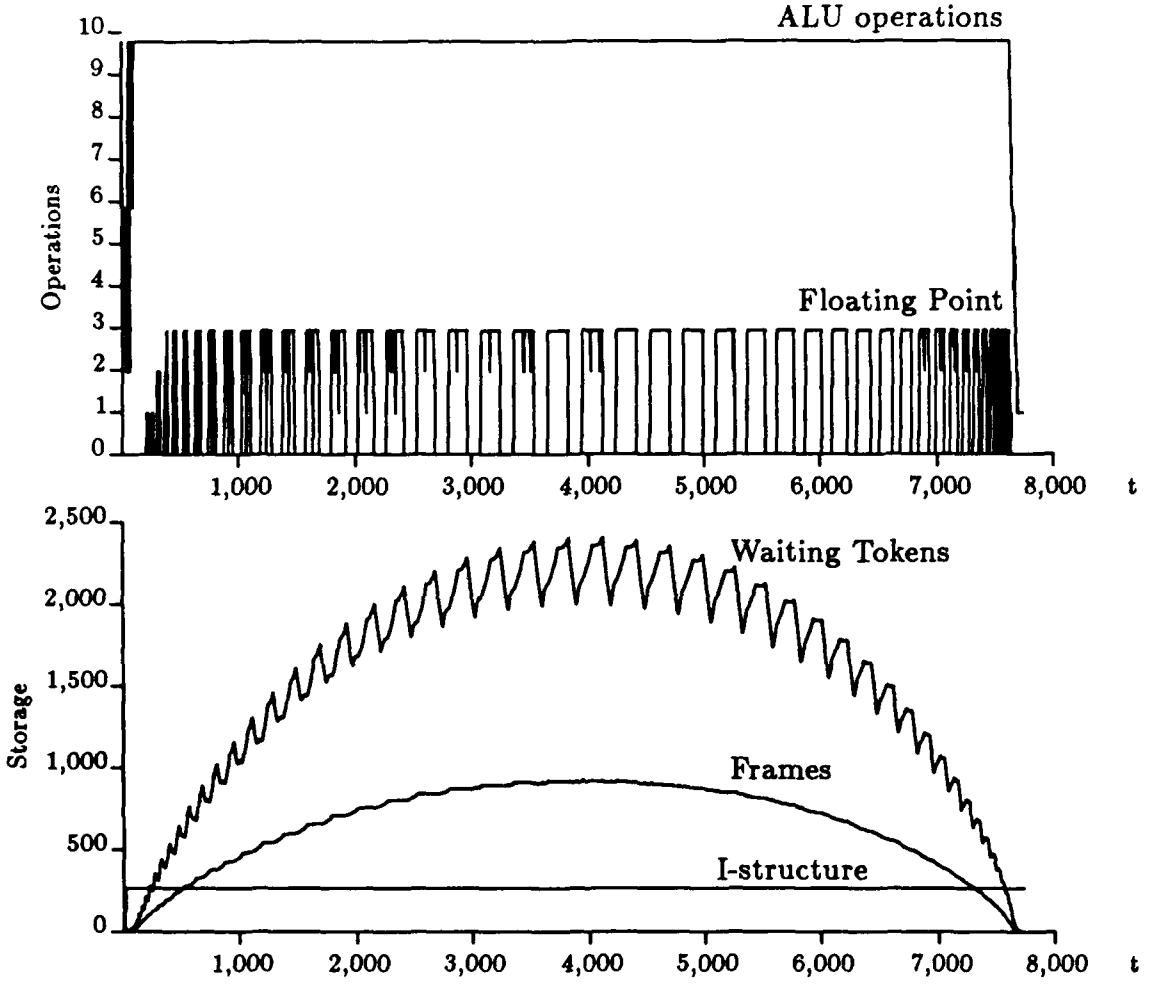


Figure 2.12: Finite-Processor Profiles ($P = 10, l = 2$) for Matrix Multiply on 16×16 Matrices

In some cases, we have seen that programs unfold in a way that causes resources to be allocated very early without contributing to reducing the critical path. In other cases, we see that programs with a reasonable ratio of resource usage to parallel activity become unreasonable when only a small fraction of the potential parallelism is exploited. In the next chapter we will see that limiting the unfolding of loops can be very effective in addressing the resource problem.

Chapter 3

k-Bounded Loops

In this chapter we present a simple mechanism for controlling the unfolding of loops in a dataflow machine. The idea is to introduce an artificial dependence between the completion of an iteration and the initiation of the one logically k later. As this allows the resources associated with the iteration to be reused, it is valid to view it as an anti-dependence in the traditional sense[64]. The basic k-bounded loop schema is described here, but details and trade-offs in graph generation are left for Appendix A. We suggest, by example, that k-bounded loops are an effective means of controlling parallelism to reduce resource requirements. This sets the stage for the bulk of the thesis, developing a systematic policy for assigning loop bounds. However, before addressing the policy issue, we demonstrate further virtues of the approach in reusing data structure storage, contexts, and entire rings of activation frames.

3.1 Basic Loop Bounding Mechanism

The starting point for our k-bounded loop schema is based on the loop schema employed under the U-interpreter[14], shown in Figure 3.1. A loop has a collection of *loop variables* defined in each iteration in terms of the values from the previous iteration; these correspond to the arcs leaving the body and feeding back into the bank of *L-switches*. All the tokens for a particular iteration are distinguished by a portion of the tag. We say a loop iteration is initiated when the first token for the iteration is produced, *i.e.*, enters the body of the loop at the output of an *L-switch*, and completes when the last token for the iteration is consumed. Between these two events the iteration is *active*. Thus, the bank of *L-switches* defines the boundary between one iteration and the next.

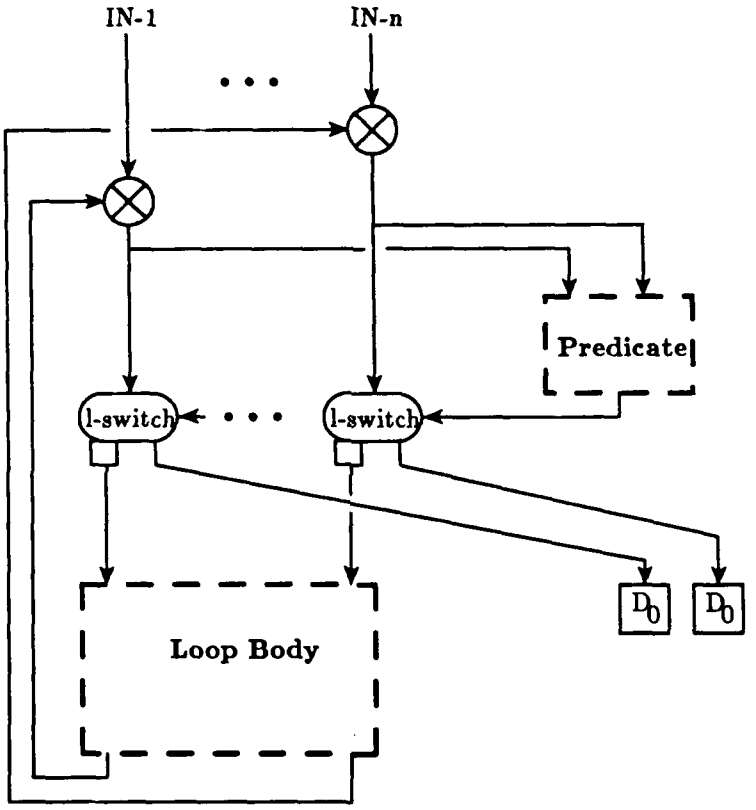


Figure 3.1: Basic Loop Schema Without k-bounding

A loop has *bounded unfolding* if, for any input values, at most a fixed number of iterations are active concurrently under *all* legal execution schedules. Note, this does not assume the ideal model or any other particular scheduling discipline; it is a statement about the graph itself. It has been shown[26] that a loop has bounded unfolding if all the loop variables are all mutually dependent and for each node in the graph there is some loop variable that depends upon the node, *i.e.*, the graph for the loop forms a single strongly connected component[3].

For example, the IP loop depicted in Figure 2.3 does not have bounded unfolding. To see this, suppose the * operation (node 11) does not fire for a suitably long period. This does not inhibit the loop variable k from circulating through the loop, as k does not depend on sum. For any value of s, eventually s iterations become active. It is enough that there exists a legal schedule that demonstrates the unbounded behavior, but here such a situation may easily arise in practice. Suppose matrix A is filled and matrix B is empty, as might occur if the producer has not progressed far enough to start filling it. Then, *I-fetches* against B will be deferred and tokens will pile up on the left input to the * node. In general, for each active iteration there will exist some number of extant tokens, although these may be represented by entries on deferred-read lists or tokens within subordinate invocations.

If a loop does not have bounded unfolding, we can cause it to have bounded unfolding by introducing artificial dependencies. Basically, we must make the initiation of an iteration depend upon the completion of the iteration logically k earlier. By definition, the loop body and predicate are well-behaved, thus when a token is produced on each output, no tokens remain in the blocks themselves. Therefore, once all the *L-switches* have fired for iteration i and produced the wave of tokens marked $i+1$, no tokens remain for iteration i. It is complete. To detect this event, we can add an output arc to each *L-switch* and reduce them to a single arc through a tree of *no-ops*, as indicated by the triangular *completion tree* in Figure 3.2. To prevent iteration i from starting, we need only inhibit the *L-switches* from firing with inputs marked $i-1$. This can be done by gating all the loop variables, but it is enough to gate the output of the predicate, since it controls all the *L-switches*. Thus, we introduce a new *L-gate* node, denoted by $\bowtie$, that receives the output of the predicate and forwards it to the *L-switches*, but only when a matching token is present on the *iteration control* arc. We require that iteration i complete before iteration $i+k$ starts, so the completion tree and the gate are connected via a D_{k-2} node, which increments the iteration number of its input by $k-2$.

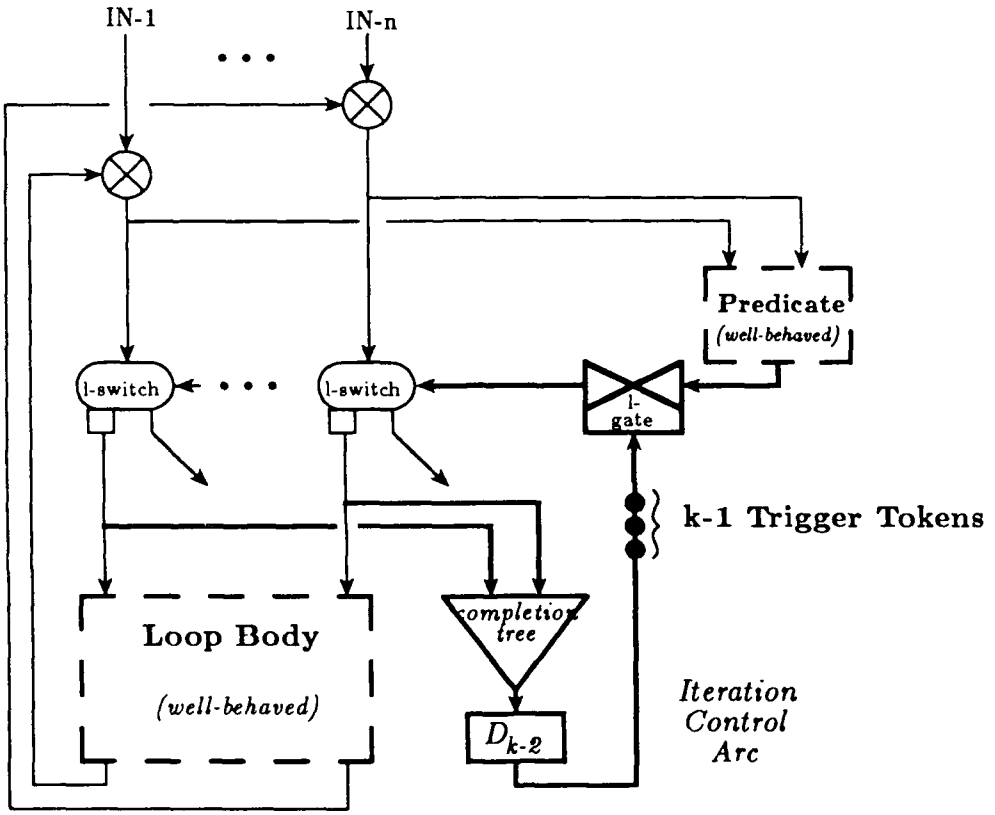


Figure 3.2: k-Bounded Loop Schema

To get the loop started, $k-1$ trigger tokens are deposited on the iteration control arc carrying iteration numbers $0,\ldots,k-2$. In Appendix A, we show that this does, in fact, allow k waves of tokens to exist in the loop body concurrently, and explain how the trigger tokens are deposited and removed. Notice that this approach requires $k \geq 2$; this phenomenon is not unlike the need for two gates in a shipping lock or two stages in a flip-flop; the appendix also describes a more costly 1-bounded loop schema. Completion detection presents an interesting trade-off: by making use of the dependencies within the loop body itself, the size of the completion tree can be reduced, however, this reduces the level of potential parallelism within the loop body for a given k.

The k-bounded loop schema is well-behaved and, therefore, may appear within another loop. We may have k_o concurrent iterations of the outer loop, which generates k_o concurrent *invocations* of the inner loop, each with some, possibly different, k_i concurrent iterations.

With this approach, the unfolding of a loop is determined dynamically at the time it is invoked. The prologue of the loop establishes k and generates the trigger tokens. Thus, a program can run with tight bounds on a small machine and looser bounds on a larger one, without recompilation. The task then becomes one of determining the proper k-bounds, given the machine configuration and the problem size.

3.2 Effectiveness of Bounded Loops

To see the effectiveness of this mechanism, we explore a couple of examples from the previous chapter. Consider first the resource behavior observed in Simple (Figure 2.11). Suppose that only two iterations of the outermost loop were allowed to execute concurrently, *i.e.*, the first must complete before the third can start, and so on. This prevents the run-away resource allocation observed previously, as can be seen in Figure 3.3[1]. There is no increase in the critical path, so whatever additional overlap occurred between outer loop iterations was, in fact, *useless*. The peak parallelism has dropped, since the burst of activity resulting from the rapid unfolding of all the iterations is prevented. The average parallelism is unchanged, however. The local

[1] Actually, all the examples shown in the thesis use the k-bounded loop schema, with unbounded execution simulated by setting the bound to the total number of iterations. The loops in Inverse_Polynomial and Region are bounded at 2 in both the Simple executions, since these are "while" loops that do not unfold beyond this amount. If a larger value were used, the activation frame usage would increase considerably, with an *increase* in τ_∞. In the Matrix Multiply example, the unfolding of the inner product loop is uniformly bounded at 3, for reasons discussed later.

storage requirement is reduced (W_{max} = 142,958, A_{max} = 29,999), but more importantly, it has become independent of the number of time-steps. This is a *qualitative* improvement, although the token storage requirement is still quite large.

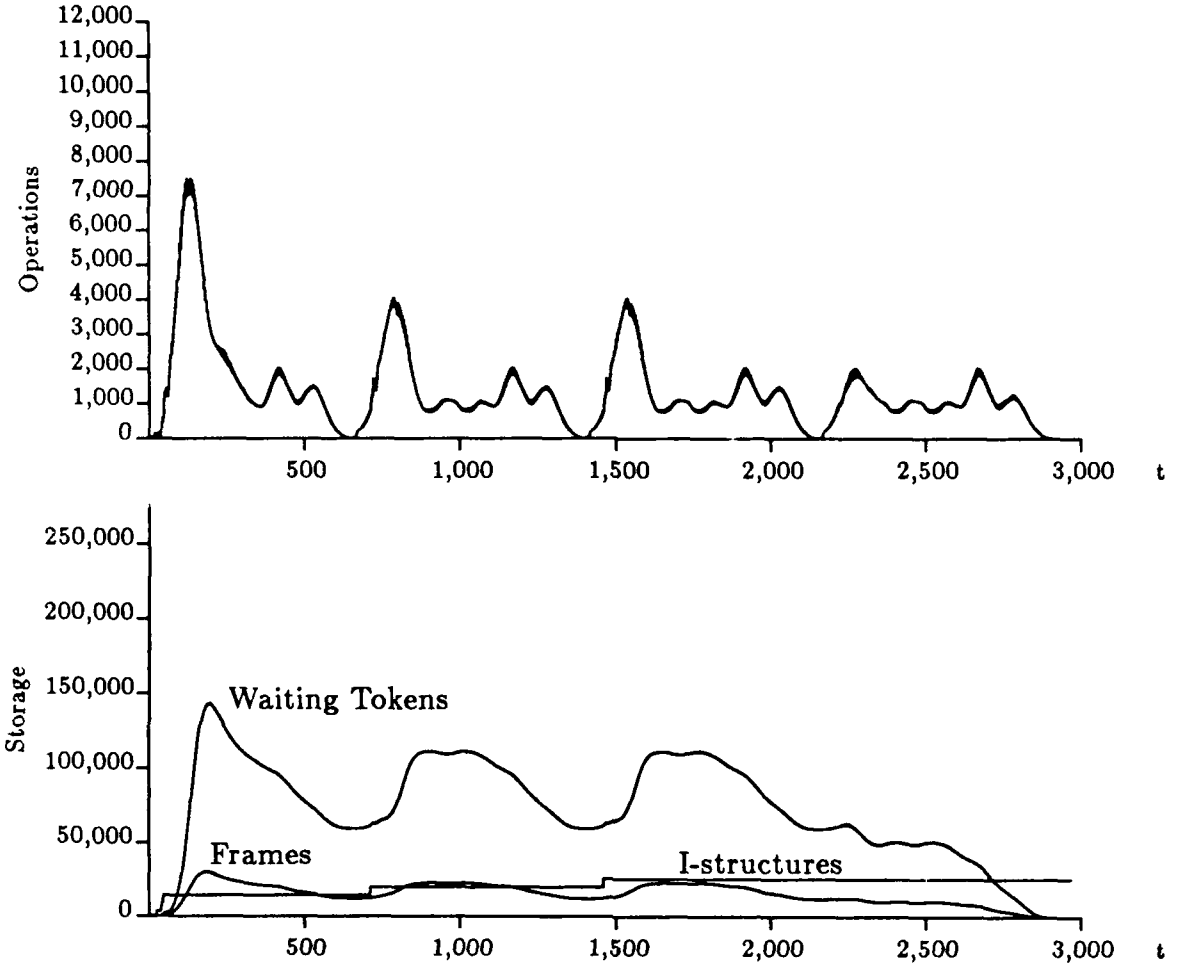


Figure 3.3: Parallelism and Resource Profiles for Simple, 16 × 16 for 4 steps, with the Outermost Loop Bounded at 2

Now consider the excessive parallelism demonstrated by the finite-processor Matrix Multiply experiment in Figure 2.12. Suppose that only four iterations of the outermost loop are allowed to execute in parallel, and for each active outer-loop iteration only four iterations of the middle loop are active. Thus, at any time at most 16 invocations of `IP` are active concurrently. With these k-bounds, the finite-processor profiles are as shown in Figure 3.4. The critical-path length, τ_∞, has increased by less than 2%, while W_{max} has dropped from 2,406 to 186. A_{max} has dropped from 922 to 67.

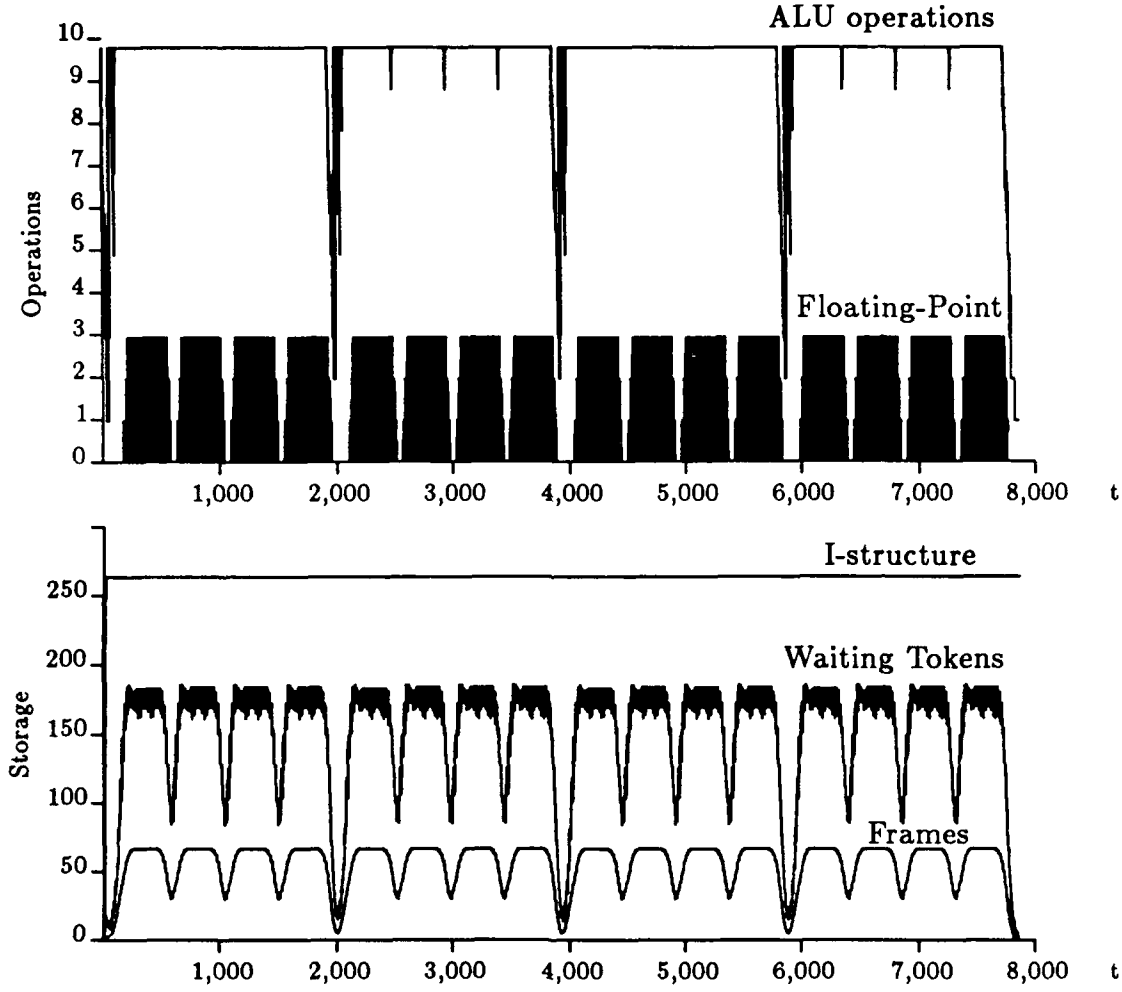


Figure 3.4: Finite-Processor Profiles ($P = 10, l = 2$) for Matrix Multiply on 16×16 Matrices with Outer and Middle Loops Bounded at 4

3.3 Reusing Resources

The k-bounded loop paradigm promotes a subtle shift in the way one thinks about tagged-token dataflow architectures. Coming from the U-interpreter viewpoint, every instruction executed in the entire lifetime of a program has a unique tag. Generation of tags, *i.e.*, allocation of tag-space, is kept simple and decentralized by using the tag-space sparsely. However, tags grow arbitrarily large. Under the bounded loop paradigm the tag-space can be used densely. Thus, tags become addresses and iterations become resources. This section explores the idea of recycling resources within the dataflow graph, in the context of k-bounded loops.

3.3.1 Recycling iterations

Since a loop can have at most k concurrent iterations and these are consecutive, *i.e.*, they are marked $i, \ldots, i + k - 1$, all tag manipulation operations can be performed modulo k. Still, all concurrent iterations will be distinguished by their iteration numbers. Thus, for a given machine, we can fix a maximum k value and thereby fix the size of the iteration field in the tag. In a manner similar to detecting completion of an iteration, completion of an invocation can be detected and its resources released[26, 85]. Thus, the context portion of the tag can be kept reasonably small, as well. The impact of these developments is that the tag-space is no longer used sparsely. This makes it possible to treat a tag as an address. The logical next step is simply to allocate a region of storage to each acyclic code-block invocation and to each iteration. Papadopoulos demonstrates an elegant development of this concept in his Explicit Token Store (ETS) dataflow architecture[74], which is being realized in the MIT Monsoon dataflow machine.

In an ETS framework, a k-bounded loop schema has a simple realization. Instead of generating a set of k trigger tokens, the loop set-up allocates k activation frames and forms them into a ring, as suggested by Figure 3.5. When an iteration completes, it sends a token to its predecessor, thereby giving the predecessor permission to send a new wave of tokens forward. The loop prologue simply builds this ring and the epilogue deallocates it. Each frame can also hold copies of loop constants and other resources, as discussed below.

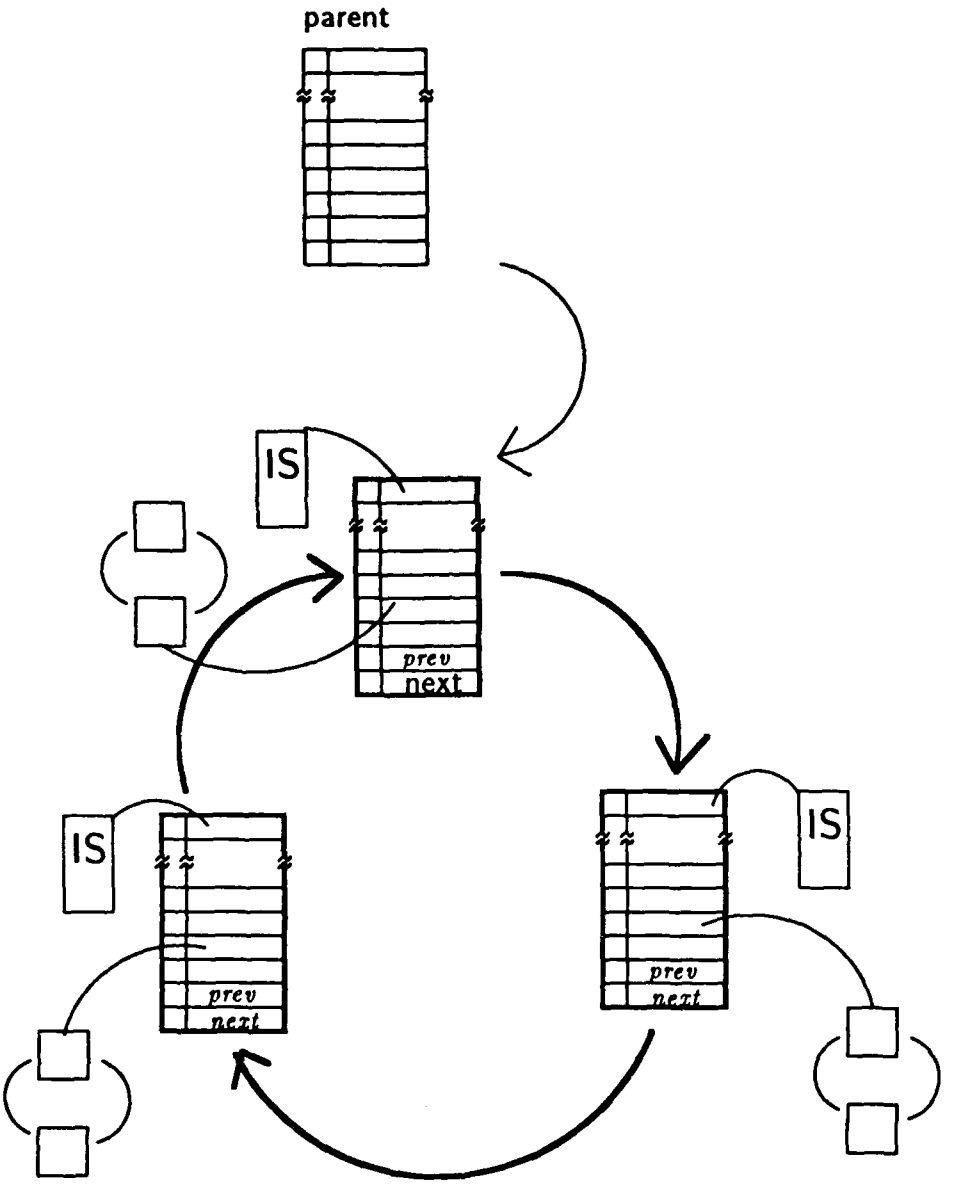


Figure 3.5: k-bounded Loop in an ETS Framework

3.3.2 Recycling data structures

A natural extension to k-bounded loops is to recycle data structures as well. Linear recurrences over large regular data structures of fixed size and shape are common in scientific applications. The `relax` program of Figure 2.7 is a prime example of such a first order recurrence; each iteration produces a new mesh using the mesh produced in the previous iteration. By bounding the outer loop to k, at most $k+1$ meshes can be in use at any time; thus, the meshes can simply be circulated within the graph. Issues related to detecting the occurrence of such a situation and clearing the meshes between uses are discussed in Appendix A.

Figure 3.6 shows ideal profiles for `relax` with the outermost loop constrained to 4 concurrent iterations, 5 versions of the mesh recycled within the graph, and innermost loops tightly constrained. This shows a qualitative difference in all of the resource profiles, but especially in the I-structure profile. We might use this k-bound assignment for a configuration capable of 25 parallel operations with a latency of 4.

Circulating data structures in this manner can be viewed as a generalization of the multi-buffering techniques used to support asynchronous I/O, but here it is used as part of the internals of the program. For an n^{th} order structure recurrence, we circulate $k+n$ structures in a k-bounded loop, where after the n^{th} use of a structure it is cleared and reused. It is possible also to allow the number of versions of the structure to be controlled independent of k, using the recycling strategy explained in Appendix A.

3.3.3 Recycling contexts

Above we noted that a k-bounded loop can appear within another k-bounded loop. Not only does this provide a great deal of freedom in the policy for controlling parallelism, it provides an opportunity for more extensive resource reuse. Suppose each invocation of the inner loop builds a ring of k_i activation frames. Each of these rings can be treated as a constant in one of the k_o activation frames associated with the outer loop. In this way, a ring of rings is formed. This storage is used heavily for the duration of the outer loop and then released in bulk. We will see that this reduces the execution overhead of the loop bounding mechanism, but also implies that a machine might exploit the locality inherent in the use of such a structure.

In addition, a deeply nested loop will usually have loop constants from several of the outer levels. By associating rings of frames with the outer loop, it is possible to avoid storing constants

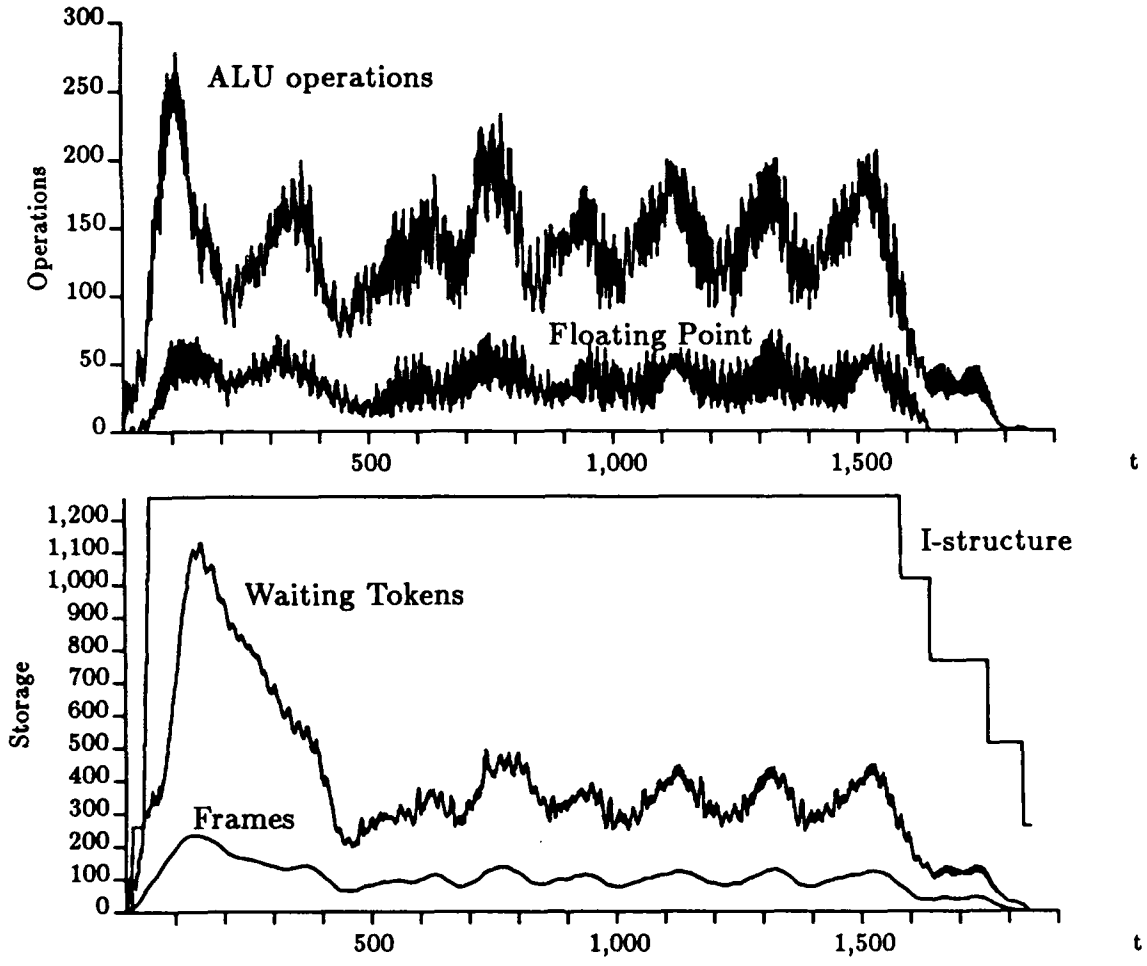


Figure 3.6: Parallelism and Resource Profiles for 16 iterations of Relax on a 16 × 16 Mesh with Loop Bounding and Structure Recycling

inherited from outer level loops multiple times. For example, our matrix multiply program has seven loop constants in the innermost loop, five of which are constant over the middle loop, as well. If the activation frame rings of the inner loop were tied to the ring for the middle loop, the five constants acquired from the outer loop need only be stored once. Each time the IP loop is activated, only the two remaining loop constants need be provided. This, in effect, combines the virtues of a multilevel environment with the access efficiency of a flat environment.

3.4 Conclusions

k-Bounded loops can be treated as a simple extension of the loop schema originally proposed for the U-interpreter and implemented quite efficiently. Signal generation is required if contexts are to be reclaimed without some external reclamation mechanism. Beyond the cost of signal generation, the number of instructions introduced to control the unfolding of a loop is equal to the number of loop variables, using the most straightforward style of completion detection. With a little graph analysis this can be reduced, but doing so can reduce the utilization of the resources assigned to a loop. The cost of constructing a 1-bounded loop in a pure dataflow context is considerably greater, although this can presumably be reduced in hybrid architectures with a more powerful mechanism for serialization[55].

The basic mechanism proves to be very effective in controlling parallelism to reduce resource requirements. This brings us to the problem addressed in the bulk of the thesis, development of a policy for assigning the k-bounds.

Chapter 4

Toward a Policy for Loop Bounding

In the preceding chapters we have seen a technique for characterizing the behavior of programs, which provided a means of demonstrating the parallelism/resource problem, and an effective mechanism for controlling parallelism. k-bounded loops provide "parallelism knobs" that can be adjusted dynamically. We turn now to the question of how to adjust the knobs. This is most naturally viewed as an optimization problem: given a resource constraint r, choose a setting of the k-bounds such that the resource usage under any execution schedule is no more than r and performance is maximized. Casting the problem in this light provides valuable insight, but has several problems. The resource usage function is complex, the space to optimize over is ill-defined, and the objective function is difficult to characterize. Even if these problems were overcome, since our target is a complex, asynchronous environment the effort placed in deriving an optimal static execution schedule is likely to be squandered. This suggests that we should relax the optimality concern and take a more *qualitative* approach: identify those loops that are clearly good candidates for exposing parallelism and those that are clearly poor candidates. Our task then becomes one of developing policies for dealing with the various loop relationships that arise and deciding where to invest resources. In this chapter, we assemble the tools we will need for formulating and testing policies in later chapters.

This qualitative approach leads to an unusual interplay of static analysis and dynamic decision making, so let us examine this interplay further. First, unlike the more traditional work on optimization and scheduling[76], we are concerned with programs in-the-large. As the program structures we deal with become larger, more crude analysis must suffice. However, as we work down into the computational kernels of the program, we need to be able to exploit more complete information. Thus, for a given program we must represent and utilize static informa-

tion at various levels of detail. Second, we cannot assume the program is to be recompiled for each problem size and machine configuration, so in most cases static analysis can only yield symbolic expressions for the size of various data structures and for the resource requirements of various portions of the program. For complex `WHILE` loops we cannot even derive a simple expression for the iteration count at compile time. On the other hand, if the program is stated in terms of simple loops over regular data structures of fixed size, we want to exploit this fact. Thus, the level of detailed revealed through static analysis will vary for different programs. In our approach, decision making is integrated with program execution and can be very dynamic. However, we try to minimize the decision making overhead by making decisions at the highest possible level. Thus, if the program is stated in terms of a fixed size data structure, we may exploit this fact by "lifting" decision making out to the beginning of the program. As the character of the program becomes more dynamic, decisions are made on a more local basis. The general rule is to make decisions at the outermost level in which the necessary parameters are known.

The very fact that decisions are made dynamically implies that they must be made quickly. We cannot afford to search a space or solve a constraint system. Therefore, at compile time we determine the parameters that enter into each decision and their inter-relationship. Based on this, we augment the program with code that *computes* loop bounds using simple formulae. A given formula embodies a *policy* for assigning k-bounds in the particular portion of the program, *e.g.*, it favors outer loop unfolding, inner loop unfolding, or some combination.

To make clear how a program is augmented with code that computes a k-bound in terms of symbolic program and resource parameters, we begin by analyzing a single loop in isolation. Then, to highlight the policy aspect of our task, *i.e.*, determining where resources should be invested, we examine the behavior of a simple nest of loops. From this examination emerges several important ideas. The first is the use of symbolic *resource expressions* to characterize the space of valid k-bound assignments for a collection of loops. A policy can then be formalized as a rule for solving a equation of the form: $R(k_1, \ldots, k_n) = r$, for each k_i "on the fly". We need not derive an accurate closed-form expression for the resource requirements of the program as a whole, however, for pieces of the program where such an estimate can be derived, we can make decisions most effectively. The second is a notion of optimality against which policies can be evaluated on simple program examples. The third is a simple model of loop unfolding that

gives a qualitative basis for policy formulation. Finally, we outline our framework for analysis and policy development, which provides the basic organization for the remaining chapters.

4.1 Determining k for a Single Loop

In this section we examine the simplest loop bounding scenario, a single loop. Although the policy issue is trivial here, it makes clear what information is extracted through static analysis and how it is used to augment a program with resource management code that computes k-bounds. Recall the 2D relaxation program in Section 2.3. In each iteration of the outermost loop, a matrix of size $(l1, u1) \times (l2, u2)$ is allocated and filled. The matrices can be recycled, as discussed in Section 3.3.2, so with k concurrent iterations of the outermost loop, $k+1$ versions of the matrix may be in use. Each iteration reads one matrix and writes the next. Thus, if this program is to execute under data structure storage constraint S, we should set the bound for the outermost loop as:

$$k = \left\lfloor \frac{S}{(u1 - l1 + 1)(u2 - l2 + 1)} \right\rfloor - 1.$$

Observe, we do not "compile in" the loop bound, k, the machine configuration, S, or even the size of the matrix. Rather, we capture the relationship between them. The program is augmented with resource management code that computes k in terms of dynamic resource parameters and program values according to the above formula. An augmented version of the **relax** program from Chapter 2 is shown below, with resource management code underlined.

```
def relax' A steps S =
  {((l1,u1),(l2,u2)) = (2D_bounds A);
   k = max steps (floor (S/((u1 - l1 + 1)*(u2 - l2 + 1)) - 1);
  in {for i <- 1 to steps bound k do
        next A = relax_step A;
      finally A}};
```

By controlling the unfolding of the loop in this manner, we ensure that the combined size of the structures in use at any time is at most S. Forming the expression for the structure size and introducing the code to compute the k-bound is straightforward. The more difficult analysis is determining that the lifetime of the structure is contained within a fixed window of iterations. In this example, the lifetime of the structure created by iteration i is contained in the lifetimes of iteration i and $i+1$, unless i is the last iteration.[1] This augmented loop could

[1] Augmenting the program in this manner is effective even if storage is reclaimed automatically through some

be used in a larger program without further analysis of its behavior, because it is *self-managing*: given only a parametric resource constraint, S, it determines its own unfolding. On the other hand, by considering the context in which this loop is invoked, we may be able to make a more educated k-bound assignment. Perhaps a small reduction in unfolding here will enable a significant amount of parallel activity elsewhere, or perhaps we may be able to optimize the resource management code if the context is known.

Although the k-bound assignment is dynamic, it does not take into account status information such as the level of activity in the machine at the time the loop is initiated. In general, given a loop and a parametric resource constraint, we analyze the program to determine a reasonably tight upper bound on the resource usage per iteration, and then we augment the program with code to determine the k-bound. If the number of iterations for a loop can be determined at the point where the loop is initiated, as in a `FOR` loop, then this can be used as an upper limit on k. Resource concerns may place further constraints on k. Observe that the k-bounded loop schema requires that a loop bound be established at the time a loop is initiated. Thus, for a loop where the number of iterations cannot be determined in advance, we either have to choose a k that is expected not to exceed the total number of iterations or compile the loop using a different schema.[2]

For a single loop there is no policy issue *per se*. We simply divide through to solve for k. However, in dealing with complex collections of loops, we must decide where to invest our resources, based on some policy.

4.2 Determining k-Bounds for a Loop Nest

To develop intuition toward the broader k-bounding problem, we consider an example with nested loops. We will look at local storage requirements, which are sensitive to the unfolding of any loop in a nest. The matrix multiply program of Section 2.3 has three nested loops, as

mechanism external to the program, since references exist to only $k+1$ structures at any time. However, the actual reclamation of structures may lag somewhat behind the completion of the corresponding iterations. Recycling structures within the loop, as discussed in Chapter 3, avoids the cost of repeated allocation and deallocation and, more importantly, guarantees that structure reuse is closely linked to completion of iterations. However, recycling structures may require transforming the program, *e.g.*, lifting the allocation of a structure up to the loop that determines its lifetime, whereas computing k-bounds only requires *augmenting* the program with new resource management code.

[2] By compiling loops essentially as tail-recursive functions it is possible to circumvent the k-bounding mechanism, however, execution costs increase and control over unfolding is lost.

indicated in Figure 4.1. Here the question is not just how to compute a particular loop bound, but where to invest resources. In Figure 2.3, we observed that for a given invocation of the innermost loop (the inner product), only a small number of iterations are active at any time. Clearly, it does not pay to invest resources there, so let us set the k-bound for this loop to a small constant, $k_i = 3$. This way we can essentially ignore the innermost loop and focus on whether we do better by investing resources in the outer loop or in the middle one.

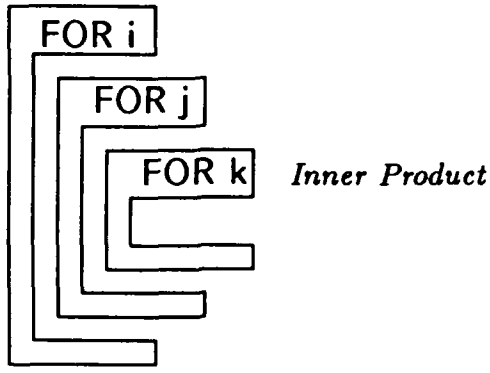


Figure 4.1: Loop Structure for Matrix Multiply

Although the mechanism allows a different k-bound to be employed for each invocation of the middle loop, there is little reason for doing so. Thus, the space of possible k-bound configurations we should consider is the cross product of the range for the outer loop unfolding parameter, k_o, and that for the middle loop, k_m. To determine where to invest resources, we need to understand what the terrain in this space is like.

k_o	k_m = 2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	.38	.52	.67	.81	.95	1.09	1.23	1.37	1.49	1.62	1.74	1.86	1.98	2.09	2.19
15	.36	.49	.63	.76	.90	1.03	1.16	1.29	1.41	1.53	1.65	1.76	1.87	1.98	2.08
14	.33	.46	.59	.71	.84	.96	1.09	1.21	1.33	1.45	1.56	1.67	1.77	1.87	1.96
13	.31	.43	.55	.66	.78	.90	1.01	1.13	1.24	1.35	1.46	1.56	1.66	1.75	1.85
12	.29	.40	.50	.61	.72	.83	.94	1.04	1.15	1.25	1.35	1.45	1.55	1.64	1.72
11	.26	.36	.46	.56	.66	.76	.86	.96	1.06	1.15	1.25	1.34	1.43	1.51	1.60
10	.24	.33	.42	.51	.60	.69	.78	.87	.96	1.05	1.14	1.23	1.31	1.39	1.46
9	.22	.30	.38	.46	.54	.62	.71	.79	.87	.95	1.03	1.11	1.18	1.26	1.33
8	.19	.27	.34	.41	.49	.56	.63	.70	.77	.85	.92	.99	1.06	1.13	1.19
7	.17	.24	.30	.36	.43	.49	.55	.62	.68	.74	.80	.87	.93	.99	1.05
6	.15	.20	.26	.31	.37	.42	.47	.53	.58	.64	.69	.75	.80	.85	.90
5	.13	.17	.22	.26	.31	.35	.40	.44	.49	.53	.58	.62	.67	.71	.76
4	.10	.14	.18	.21	.25	.28	.32	.36	.39	.43	.47	.50	.54	.57	.61
3	.08	.11	.14	.16	.19	.22	.24	.27	.30	.32	.35	.38	.41	.43	.46
2	.06	.08	.09	.11	.13	.15	.17	.18	.20	.22	.24	.25	.27	.29	.31

Figure 4.2: W_{max} Over Entire k-Space for 16 × 16 Matrix Multiply in Units of 1,000 Tokens

One way to build this understanding is to enumerate the entire space and examine the

resulting surface. This is clearly unrealistic for large programs, but is applicable to simple programs over a reasonable range of k-bounds. Figure 4.2 gives the maximum local storage requirement W_{max}(in units of 1,000 Waiting Tokens), obtained by executing the program under the ideal model for every point in the entire k-space for Matrix Multiply on 16 × 16 matrices, with the unfolding of the innermost loop fixed at three. The first observation is that the resource requirement increases monotonically with the k-bounds. The second is that the surface is not symmetric: the waiting token requirement increases more rapidly with respect to outer loop unfolding than with respect to middle loop unfolding. Since the bulk of the work is performed within the inner product, we may consider various k-bound configurations that allow the same number of concurrent invocations of the inner product. In k-configuration $(k_o, k_m) = (16, 2)$ the requirement is 380 tokens, while at (2,16) it is only 310. Intermediate configurations which also allow 32 concurrent invocations of the inner product exhibit requirements between these values, 340 at $(8, 4)$ and 320 at $(4, 8)$.

Figure 4.3 gives the average parallelism, P_{ave}, measured under the ideal execution model over this same region. Like token storage requirements, P_{ave} increases monotonically with the k-bounds, although the surface is not as smooth as that for W_{max}. Observe the dramatic increase in P_{ave} from $k_o = 15$ to $k_o = 16$, whereas Figure 4.2 shows no corresponding jump. The P_{ave} surface also is not symmetric; it increases more rapidly with unfolding in the outer loop. The same configurations mentioned above exhibit P_{ave} of 84, 83, 74, and 52, respectively, by increasing k_m. This is encouraging, for if parallelism is proportional to resources everywhere, locating a good k-bound configuration is fairly simple. However, we should look at the situation here in more detail.

For a given resource constraint there corresponds a *feasible region* of k-configurations that allow the program to execute without requiring more than the specified amount of resources. Figure 4.4 indicates the feasible region for our example with a limit of 350 tokens. We can ask which feasible configuration yields the best execution time, or maximum P_{ave}. To see this, we can examine the measured P_{ave} over the same region, as depicted in Figure 4.5. For this particular resource limit we see that the optimal k-bound configuration has the outer loop bound set to 8 and the middle loop bound set to 4. The sequence of configurations defined in this way for increasing resource constraints define a curve of optimal configurations, a k-curve, which we would like to capture in a simple policy.

k_o	k_m = 2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	[84]	108	147	152	183	187	228	232	235	238	242	245	249	254	287
15	46	61	86	89	111	114	147	150	153	155	158	161	164	169	200
14	45	60	85	89	110	113	145	148	151	154	156	159	162	167	196
13	45	60	85	88	110	113	144	147	150	152	155	158	161	165	194
12	45	60	85	88	109	112	143	146	148	151	154	157	159	164	192
11	45	60	84	87	109	111	142	145	147	150	153	155	158	162	190
10	45	60	84	87	108	111	141	144	146	149	151	154	157	161	188
9	45	59	83	86	107	110	140	143	145	147	150	153	155	160	187
8	45	59	[83]	86	106	109	138	141	144	146	149	151	154	158	184
7	30	41	58	60	75	77	98	101	102	104	106	108	110	113	134
6	30	40	57	59	74	76	97	100	101	103	105	107	109	112	132
5	23	31	44	45	57	58	75	77	78	79	81	83	84	87	103
4	23	30	43	45	56	57	[74]	75	77	78	80	81	83	85	101
3	15	20	29	30	38	39	50	51	52	53	54	55	57	58	69
2	12	15	22	23	28	29	38	39	39	40	41	42	42	44	[52]

Figure 4.3: P_{ave} Over Entire k-Space for 16 × 16 Matrix Multiply

k_o	k_m = 2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16															
15															
14	.33														
13	.31														
12	.29														
11	.26														
10	.24														
9	.22	.30													
8	.19	.27	.34												
7	.17	.24	.30												
6	.15	.20	.26	.31											
5	.13	.17	.22	.26	.31	.35									
4	.10	.14	.18	.21	.25	.28	.32								
3	.08	.11	.14	.16	.19	.22	.24	.27	.30	.32	.35				
2	.06	.08	.09	.11	.13	.15	.17	.18	.20	.22	.24	.25	.27	.29	.31

Figure 4.4: W_{max} Feasible Region for Limit of 350 Tokens for 16 × 16 Matrix Multiply

Given complete information regarding the resource requirements and potential parallelism at all possible k-bound configurations, we could search for the optimal configuration under any resource constraint. Of course, obtaining this information in the manner presented here would be out of the question in practice. Running the program even once to deduce its properties is somewhat unsatisfactory, although several researchers advocate just that[41, 81], but running it for many possible configurations is absurd. In the case study presented later we consider a program containing roughly 300 loops, nested six levels deep. The space of possibilities is immense! Even for simple programs, the size and shape of the space may be highly data dependent. More importantly, optimality under particular scheduling assumptions is at best a guideline. The resource requirements exhibited under a particular execution schedule, *e.g.*, the ideal model, are not necessarily the worst case, so the configuration chosen in this way may deadlock under different scheduling assumptions. Finally, the parallelism exhibited on a real machine will be influenced by a variety of other factors.

k_o	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16															
15															
14	45														
13	45														
12	45														
11	45														
10	45														
9	45	59													
8	45	59	**83**												
7	30	41	58												
6	30	40	57	59											
5	23	31	44	45	57	58									
4	23	30	43	45	56	57	74								
3	15	20	29	30	38	39	50	51	52	53	54				
2	12	15	22	23	28	29	38	39	39	40	41	42	42	44	52

k_m

Figure 4.5: P_{ave} for 350-Token Feasible Region for 16 × 16 Matrix Multiply

The real point of the discussion is that it establishes a *modus operandi* for attacking the problem of developing policies for loop bounding. We can view the ideal model as representing the "expected case," in the sense that a k-bound assignment that achieves near optimal performance under that model is likely to be a reasonably good assignment for a realistic machine. Thus, this rather computationally intensive optimization framework provides a metric

for evaluating various policies on particular small examples. We can ask, for a given resource constraint, what fraction of the maximum achievable parallelism does a given policy achieve. This allows us to make stronger assertions than would be possible by only comparing policies with one another.

Furthermore, it is clear that we cannot back away from the feasibility issue. If we are to assign k-bounds in a manner that ensures the program will stay within some resource constraint, we must characterize the feasible region in some manner. We must be able to estimate the resource requirements for a given k-bound configuration or possibly even give a closed-form expression for the requirement, in terms of the k-bounds and certain program variables. This issue is addressed in detail in the next section. Figure 4.2 provides grounds for optimism in this effort, because in scanning over it we see that the change in the resource requirement is fairly smooth.

Given the means of estimating resources to ensure feasibility, we still need to choose a near-optimal k-bound configuration rapidly. Although deriving a meaningful closed-form estimate of parallelism appears to be very difficult, there is still cause for optimism. Following the method of evaluation suggested above, let us define a k-bound configuration as *good* if it exhibits maximal P_{ave} under the ideal execution model for some resource limit. The configurations marked by boxes in Figure 4.6 are good in this sense. Thus, most points in the lower left quarter are good, and beyond that the outer loop should be allowed to unfold completely. If this characterization could be established at compile time, it would not be difficult to arrive at a formula for computing the k-bounds, *i.e.*, a policy for choosing a configuration, that stays within these good regions. A simple analytic model of loop unfolding is presented later in this chapter as a basis for such a characterization.

4.3 Estimating Resource Requirements

In this section we focus on the problem of estimating the resource requirements. The idea is to form a simple symbolic expression for the maximum resource requirement of a program or subprogram in terms of k-bounds for the constituent loops[6]. Such an expression provides a conservative characterization of the feasible region for any given resource constraint. We begin with the requirements for individual code-block invocations and iterations and then extend this

k_o															
16	84	108	147	152	183	187	228	232	235	238	242	245	249	254	287
15	46	61	86	89	111	114	147	150	153	155	158	161	164	169	200
14	45	60	85	89	110	113	145	148	151	154	156	159	162	167	196
13	45	60	85	88	110	113	144	147	150	152	155	158	161	165	194
12	45	60	85	88	109	112	143	146	148	151	154	157	159	164	192
11	45	60	84	87	109	111	142	145	147	150	153	155	158	162	190
10	45	60	84	87	108	111	141	144	146	149	151	154	157	161	188
9	45	59	83	86	107	110	140	143	145	147	150	153	155	160	187
8	45	59	83	86	106	109	138	141	144	146	149	151	154	158	184
7	30	41	58	60	75	77	98	101	102	104	106	108	110	113	134
6	30	40	57	59	74	76	97	100	101	103	105	107	109	112	132
5	23	31	44	45	57	58	75	77	78	79	81	83	84	87	103
4	23	30	43	45	56	57	74	75	77	78	80	81	83	85	101
3	15	20	29	30	38	39	50	51	52	53	54	55	57	58	69
2	12	15	22	23	28	29	38	39	39	40	41	42	42	44	52
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

k_m

Figure 4.6: "Good" Points in k-Space for 16 × 16 Matrix Multiply, shown overlaying P_{ave}

analysis in terms of the two rules of composition for loop programs: nesting and adjacency. Two forms of local storage are considered, tokens and activation frames, as well as I-structure storage. In the process of developing these resource expressions certain key policy parameters emerge.

4.3.1 Individual code-blocks and loops

We begin by examining the token storage requirements of individual acyclic graphs and individual loops. A very crude estimate for the token storage requirement of an acyclic graph is the number of arcs, since each token corresponds to some arc. If we are only concerned with waiting tokens, we could count the number of nodes with two inputs. In either case, the result is likely to be unsatisfactory because no execution schedule obtains the estimate. We have ignored the dependencies implied by the graph itself. On the other hand, we could compute some kind of "typical" token requirement by dividing the number of nodes by the critical path, but this is unsatisfactory because many legal execution schedules may exceed the requirement, raising the possibility of deadlock. The number we would like to determine is the maximum resource requirement over any legal execution schedule.

This worst case resource requirement can be expressed as the maximum cut through the graph. Although max-cut is NP-complete in general[42], for the restricted class of graphs

we consider it can be reduced to the dual of a min-cost flow problem, and thereby solved in polynomial time[73]. This approach is pursued in Appendix B, where we form a linear program representing the token storage requirement of a code-block invocation[26]. In applying this technique to real *Id* applications we find that the linear programs are quite large, however, some straightforward optimizations make the approach practical. The estimates obtained in this fashion may be somewhat loose for any particular machine, but are achieved by some, perhaps unlikely, execution schedule.

In the presence of conditionals we either must settle for a somewhat pessimistic estimate, *i.e.*, one that may not be achieved by any execution schedule, or consider all possible settings of the conditionals, using a branch-and-bound technique to limit the search. I-structure references may also cause the estimate to be pessimistic, since the *I-store* and *I-fetch* for a given element represent a data dependence that is not apparent in the graph, except possibly via subscript analysis[88].

The requirement for a k-bounded loop can be estimated as k times the requirement of the graph obtained by breaking the cycles in the loop at the *L-switches*. However, this gives an overly pessimistic estimate, since some values must circulate in order for the predicate to be evaluated and the loop to unfold. The constraint system technique in Appendix B takes this into account and produces a tight estimate of the form $t_1 + kt_2$. For example, the graph for the IP loop, shown in Figure 2.3 has 13 nodes and 19 arcs, including the D_{k-2} node introduced to implement the k-bound between the two *L-switches* and the *L-gate*. If we break the cycle, the max cut contains 7 arcs. The constraint system technique gives an estimate of $1 + 4k$, substantially better than $7k$. The ideal profiles in Figure 2.3 show a maximum of 6 tokens, with at most two concurrent iterations. This does not include tokens on the iteration control arc. Thus, for this example the constraint-system estimate is quite good.

As mentioned above, even if a resource estimate is achievable under some execution schedule, it may be overly pessimistic for any realistic machine. For example, in Chapter 3 we observed that all but a very special class of loops have potentially unlimited unfolding. Even our Inner Product example may unfold to any number of concurrent iterations under a manifestly unfair execution schedule, yet we saw in Figure 2.3 that under the ideal execution model only two iterations were active at any time. We will see below that many loops have such a natural unfolding limit, with certain assumptions regarding fairness and communication latency. As

long as our loop bounding policy does not assign k-bounds far in excess of the natural unfolding, resource estimates of the above form can be reasonably accurate, since all k of the iterations should be active concurrently. Of course, it is unlikely that all active iterations will be in the worst case token storage configuration simultaneously, as assumed in the estimate, so the estimate is still conservative.

In considering the number of activation frames, the situation is simpler: each potentially active loop iteration is allocated an activation frame of fixed size. In effect, the variation in the token configuration over time is addressed by the compiler in the assignment of frame slots. The size of the frame reflects the worst case load. We will generally simplify things further and assume frames are of uniform size, so we only need to account for the number of frames. Because a frame is allocated for an invocation before the argument values are even available to be sent to the invocation, frame requirements observed under realistic execution schedules are often close to worst case.

For structure storage, the size of the structure must be determined through program analysis, as discussed in Section 4.1. Basically, we locate the bounds expressions at the point of allocation and trace back through the graph to produce a simple symbolic expression for the structure size. The size is independent of k-bounding. We are restricting our attention to programs in which the lifetimes of structures are known, *i.e.*, can be tied to a particular invocation/termination events. The lifetime of all structures can be tied to the top-level invocation, but that misses the point. To recirculate structures as described in Chapter 3, the lifetimes of all but the first and last versions of the structure must be tied to a window of iterations of the loop forming the structure recurrence. By limiting the unfolding of that loop, we limit how many versions of the structure co-exist. This will not be sensitive to variations in the execution order, and hence the I-structure storage requirement of a loop with a structure recurrence is linear in the k-bound.

4.3.2 Nested loops

For nested loops, the total resource requirement is linear in each of the k-bounds. Consider a loop with bound k_1 containing a loop with bound k_2. Each iteration of the outer loop gives rise to k_2 concurrent iterations of the inner loop, giving a resource requirement of the form $r_1 + r_2 k_2$. Since there may be k_1 concurrent iterations of the outer loop, the total resource requirement is

of the form

$$R(k_1, k_2) = r_1 + k_1(r_2 + r_3 k_2).$$

For activation frames it is trivial to compute the resource coefficients.[3] For token storage, the constraint system technique can be applied on each code-block to determine the coefficients. The presence of conditionals in the outer loop may imply that not every iteration invokes the subordinate loop. With detailed analysis of the conditional it may be possible to improve the estimate. Otherwise, we must assume the worst case.

For Matrix Multiply, with the k-bound of the inner product set to 3, the token storage analysis technique presented in Appendix B gives an estimate of

$$W_{est}(k_o, k_m) = 1 + 23k_o + 32k_o k_m.$$

This is an estimate of total token storage, not just waiting tokens. The estimate is somewhat loose, because it does not recognize the strictness of the innermost loop, and could be improved with more detailed analysis.

The expression for activation frames is

$$A(k_o, k_m) = k_o + 4k_o k_m,$$

since the bound for the IP loop is fixed at 3. This is the worst case under all execution orders and also fits the activation frame requirements measured under the ideal model very closely.

4.3.3 Adjacent loops

So far we have considered only perfect nests of loops, but the situation becomes more complicated as we consider more involved programs. For example, the loop structure for the relaxation example is shown in Figure 4.7. Here we have five loops: an outer loop containing three loops, one of which contains an additional loop. We call loops at the same loop nesting level *adjacent* loops.

For a collection of adjacent loops, the resource requirement is essentially the sum of the resource requirements of the individual loop nests. Note, this assumes that any two loop nests execute concurrently and exhibit their worst case resource load simultaneously. If they could

[3] These resource expressions are analogous to the determination of the number of virtual processors required in Cytron's DOacross framework[29].

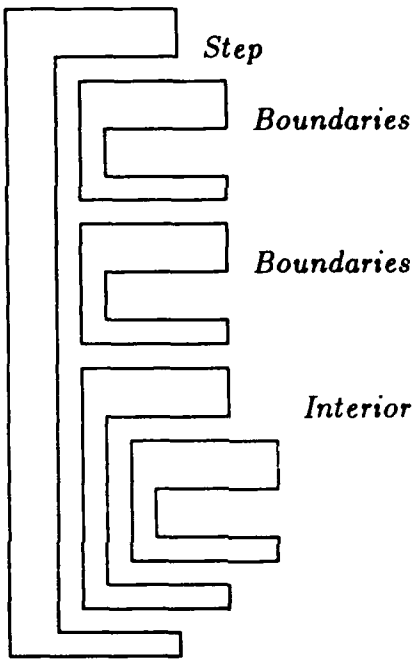


Figure 4.7: Loop Structure for Relax

be shown to execute serially, the estimate could be reduced to the maximum of the individual requirements, but this situation is rare in practice and, in general, two loops can not even be forced to execute serially[86]. The more common situation is that adjacent loops overlap in some fashion, and the nature of the overlap can be quite subtle. Determining when the peak combined load is provably less than the sum of the peak loads is an interesting problem, requiring involved dependence analysis, but is not pursued here.

4.3.4 Policy definition

We can now define more precisely what we have been calling a "loop bounding policy." Let the worst case resource requirements of a program be characterized by a *resource expression*, $R(k_1, \ldots, k_m)$. This is derived by analysis of the program, and must provide a conservative characterization of the feasible region for any resource limit, r. For any given r, there are a large number of valid k-configurations. A *resource-limited loop bounding policy* is a *rule* for computing the various k_i, given a resource limit r.

To make this notion more concrete, consider once again the Matrix Multiply example. We intimated the first policy decision for this program in the previous section: the innermost loop allows only a small number of concurrent iterations. This was based on examination of the

parallelism profile in Figure 2.3, however, a more formal argument to this effect appears below. The figure indicates that only two iterations are active at any time, however, with the additional arcs for k-bounding, not shown in the figure, that number increases to three. A policy that allows the innermost to unfold to this extent corresponds to the rule $k_i = 3$. We have solved for one k-bound parameter trivially. Taking this for k_i, we have the activation frame expression derived above:

$$A(k_o, k_m) = k_o(1 + 4k_m).$$

Now given a resource constraint, r, we must choose k_o and k_m such that $k_o(1 + 4k_m) \leq r$ and P_{ave} is close to maximal. Following a policy favoring outer loop unfolding, we first assume $k_m = 2$ and solve for k_o, using the number of iterations of the outer loop, N_o, as a maximum value:

$$k_o = \min(N_o, \left\lfloor \frac{r}{9} \right\rfloor).$$

If $k_o = N_o$, we use this value for k_o and solve for k_m:

$$k_m = \min(N_m, \left\lfloor \frac{r}{4k_o} - \frac{1}{4} \right\rfloor).$$

The policy embodied in this rule may or may not be a good choice for this program; we need to develop an understanding of the dynamic behavior of loop programs, based on their structure, that will guide in choosing the policy for a given loop or loop nest. The policy defines the general form of a mapping from resource constraints to k-bound configurations; the resource expression determines the detail.

4.4 Estimating Parallelism

In this section we address the other side of the policy question, characterizing the potential parallelism in loops. We develop a simple model of loop unfolding in the absence of k-bounding, and then extend this model to include the effects of the k-bounds. An important concept that emerges from this is the maximum useful unfolding of a loop; where present, this gives a practical upper limit for the k-bound. Also, we discover several factors that reduce the potential parallelism in a loop; these are investigated further in the next chapter.

4.4.1 Simple model of loop unfolding

To characterize the potential parallelism in loops, we develop a very simple model of loop unfolding, which essentially treats loops as a kind of "soft pipeline." We saw in Chapter 2 that a loop unfolds when the loop variables that control the predicate spawn off computations that lag behind. These two aspects of loop structure, the rate at which variables circulate and the amount of computation that lags behind, play a central role in our model.

To introduce some of the concepts we consider once again the IP example shown in Figure 2.3. From the parallelism profile we see that a new iteration is initiated every four steps and each iteration has a duration of eight steps. Thus, at most two iterations can be active at a time. The variable **k** controls the loop predicate, and a cycle of four steps is required to initiate an iteration: *L-switch*, *+1*, *<=*, *L-gate*. Each time around, tokens are fed into the portion of the graph that performs the fetch-multiply-add, allowing that portion to execute while the next iteration begins. The duration of an iteration is the length of the longest simple path from the output of the predicate to the data input of an *L-switch*. A total of twelve operations are executed in each iteration.

Our simple model of loop behavior characterizes a loop in terms of three structural parameters. Each iteration involves executing a certain number of operations, o, over some duration, c, and iterations are staggered by some amount, i. These parameters are clear in our IP example. They essentially describe the shape of the graph forming the loop body, namely, the total size of the graph (o), the length of the cycle that spawns iterations (i), and the length of the critical path through the body (c). Thus, by definition, $i \leq c$. The behavior of a loop can be visualized as in the top portion of Figure 4.8, where each iteration is depicted as a rectangle of area o and length c. The height is the average parallelism per iteration, $\rho = o/c$. This is a rather primitive characterization, as the actual profile for an iteration will have an uneven contour, but it will serve our needs. Since iterations are staggered by i, the number of concurrent iterations grows for $\lceil c/i \rceil$ iterations, but by the time the next one starts the first iteration has finished, and similarly for the remaining iterations. The loop reaches a steady state with at most $\lceil c/i \rceil$ concurrent iterations. This number is quite important, so we will give it a name. We call $\mu = \lceil c/i \rceil$ the *maximum useful unfolding* of a loop.[4] In k-bounding a loop, we will want to avoid setting

[4] In deriving μ from the graph of a program we will have to make certain assumptions about latency and scheduling. The assumptions we will use are zero latency and ideal scheduling, so we might call this μ^0. It would be interesting to consider μ^l as well, for communication latency l, but we do not pursue that line of study.

k larger than μ.

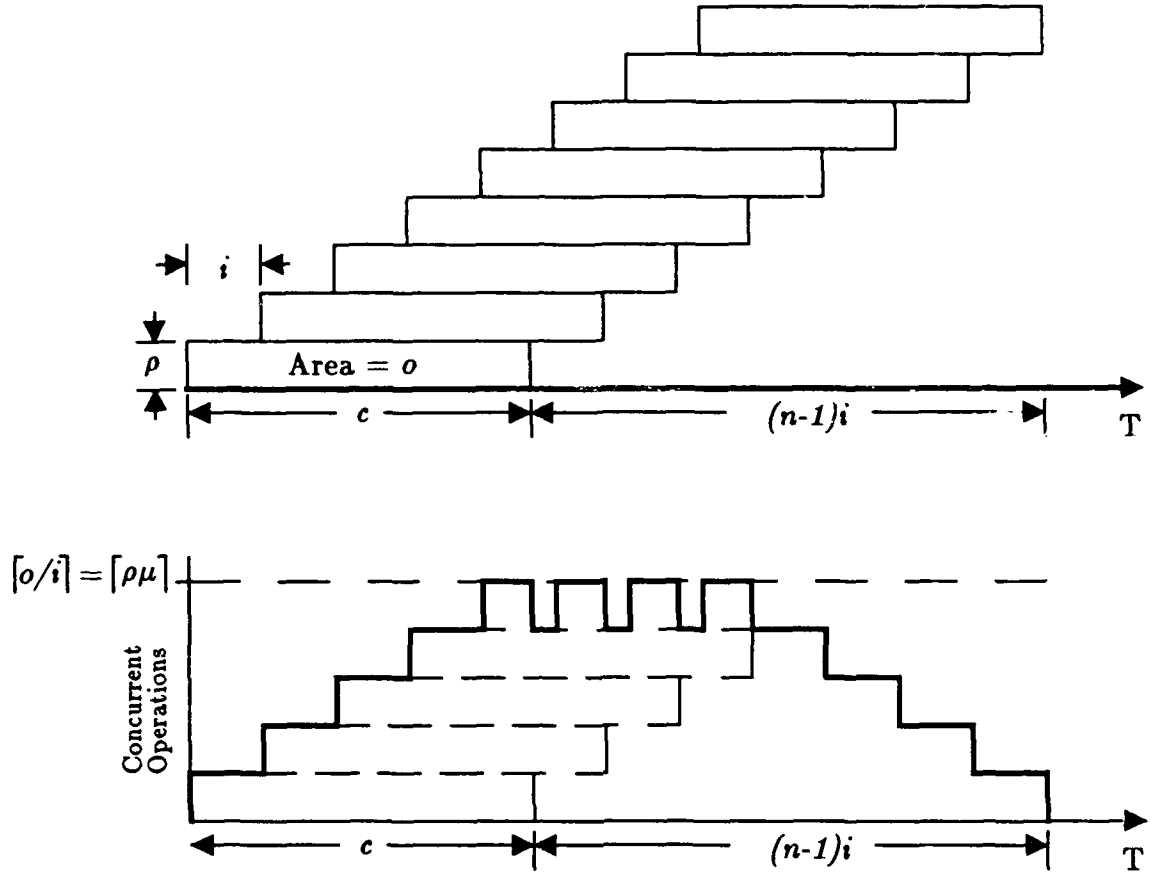


Figure 4.8: Simple Model of Loop Unfolding

Based on this model, we can derive the parallelism profile for a loop invocation by counting the number of rectangles stacked above each point, as shown in the lower portion of Figure 4.8. We can see that it takes some time to "fill the pipe" initially and to flush it at the end, giving the profile a trapezoidal shape.[5] We can see that in steady state we have an average of c/i rectangles of height o/c, so the asymptotic average parallelism is $o/i = \rho\mu$. Due to the "fill and flush" portions, the actual average parallelism will fall somewhat below this mark.

We can formalize this argument as follows. The critical path for n iterations of a loop is the time to reach the last iteration plus the duration of that iteration, as indicated in Figure 4.8. Drawing upon the notation introduced in Chapter 2, we can express this as:

$$\tau_\infty(n) = (n-1)i + c \tag{4.1}$$

[5]For nested loops, modeling the iterations as rectangles is somewhat defective, as they should be trapezoids. In which case the profile for the nest is a convolution of trapezoids, which gives the kind of bell shape curve we see for Matrix Multiply in Figure 2.4.

(Notice, this is essentially the formula describing execution time for an ideal pipeline, where c is the time required to get into steady state[63]. It is equivalent to the formula derived by Cytron in modeling DOacross loops[28], where i is the delay per iteration.[6]) Using (4.1) the average parallelism of the entire loop invocations is:

$$\begin{aligned} P_{ave}(n) &\triangleq \frac{\tau_1(n)}{\tau_\infty(n)} \\ &= \frac{no}{(n-1)i+c} \\ &= \frac{\rho\mu}{1+(\mu-1)/n} \end{aligned} \tag{4.2}$$

Thus, the asymptotic parallelism, for large n, is $\rho\mu$, the average parallelism per iteration multiplied by the number of concurrent iterations in steady state, which certainly makes sense. Using (4.2), we see that at $n = \mu$ essentially half of this value is obtained, and the increase is slow from that point on, as plotted in Figure 4.9.

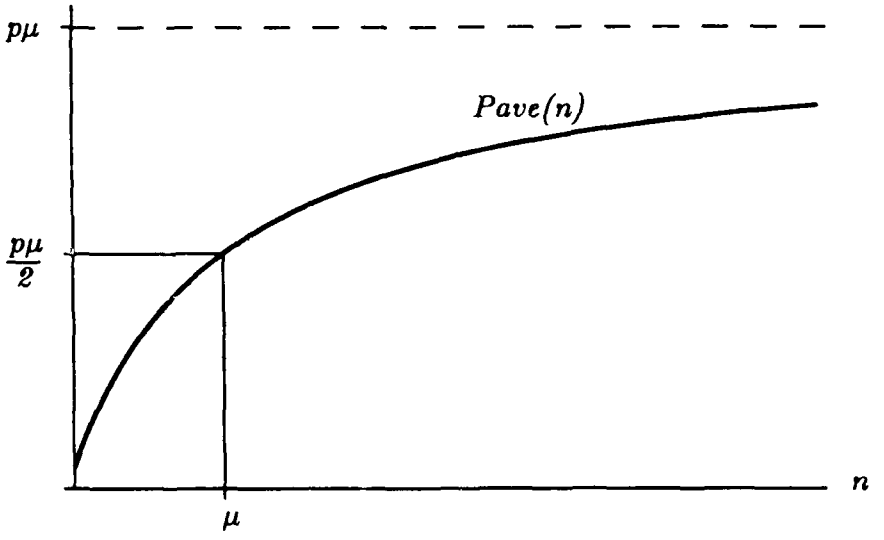


Figure 4.9: $P_{ave}(n)$ According to Our Simple Model

One important conclusion from this model is that for a loop where μ is known, it makes little sense to consider k-bounds larger than μ. Our IP example is a case in point. The more interesting situation is where $n < \mu$. In this case the loop has the potential to unfold completely, and we can examine its behavior as k approaches n. We take up this point below, after verifying our model against an example program.

[6] One difference is that here i is the time required for data to first become available for an iteration, whereas in the DOacross case the delay is determined by when the last data passed between iterations becomes available.

4.4.2 Validation of the model

To validate the model, we examine how it predicts the behavior of Matrix Multiply. Analyzing the graph produced by the *Id* compiler for IP we find $o_{IP} = 13$, $i_{IP} = 4$, and $c_{IP} = 9$. Thus, we have $\mu = 2.25$ and $\rho = 3.25$ operations per step. The extra operation and extra step in the critical path beyond what appears in Figure 2.3 are due to the D_{k-2} node in the completion tree, which connects to the *L-gate*, as discussed in Chapter 3. Even though the program is compiled using the k-bounded loop schema, we can simulate unbounded execution by using sufficiently large k-bounds.

Empirical measurements with several values of n show that the critical path length for n iterations at $k = 3$ is $4n + 24$, while at $k = 2$ the critical path length increases to $4.5n + 20$ and at $k = 4$ it is $4n + 28$. This confirms that the maximum unfolding under the ideal model is indeed three. The change in the constant term is the number of steps required to clean up each additional trigger token. Although we have not discussed the dynamic behavior of loops once the k-bounds take effect, these measurements suggest that the initiation rate is the larger of the length of the cycle controlling the predicate, i, and c/k.

The critical path measurements also indicate that the prologue and epilogue of the loop introduce approximately 20 steps on the critical path. The total operations for an invocation of IP is measured to be $13n + 53$, so there about 50 operations outside the loop. We will take these last two numbers as typical and use them for the other loops as well.

The graph for the body of the middle loop, excluding prologue and epilogue, has 27 nodes, with a longest path of 10 nodes and a predicate cycle of length 4. Thus, for a single iteration of the middle loop, including n iterations of the inner loop, we have

$$\begin{aligned} i_m &= 4 \\ o_m(n) &= 13n + 53 + 27, \ (288, \text{ at } n = 16) \\ c_m(n) &\approx 4n + 24 + 10, \ (98, \text{ at } n = 16). \end{aligned}$$

With $n = 16$ we have $\rho_m = 2.94$ and $\mu_m = 24.5$. For the fully unfolded case, the parallelism for an invocation of the middle loop is

$$P_{ave} \approx 72 \frac{1}{1 + 1.53} \approx 28.$$

The graph for the body of outer loop has 29 nodes, with a longest path of 10 and predicate

cycle of 4. We assume the prologue and epilogue of the middle loop involve 50 nodes, with a critical path of 20, as for the IP loop. Thus, we have

$$\begin{aligned} i_o &= 4 \\ o_o(n) &= (13n + 80)n + 50 + 29, \ (4687, \text{ at } n = 16) \\ c_o(n) &\approx 4n + (4n + 34) + 10 + 20, \ (192, \text{ at } n = 16) \end{aligned}$$

With $n = 16$ we get $p_o = 24.4$ and $\mu_o = 48$.

$$P_{ave} \approx 1172 \frac{1}{1 + 3.0} \approx 293.$$

This is reasonably close to the observed number of 287, and is expectedly on the high side, since we did not include some of the outer level setup and termination. Note that without accounting for the fill-and-flush behavior of loop unfolding, we would have arrived at an estimate of $256 \times 3.25 = 832$. As it is, we get a reasonably accurate characterization of the bell-shaped parallelism profile of Figure 2.4.

4.4.3 Behavior of k-bounded loops

Now we embellish our model to characterize the the behavior of k-bounded loops, where $k \leq \mu$. (When the k-bound is greater than the maximum unfolding, the behavior is as in the simple model above, although from a policy viewpoint we will want to avoid such situations.) The first k iterations are initiated in a staggered fashion, just as under the simple model. The $(k+1)^{th}$ iteration is delayed until the first iteration finishes, as indicated by the hatched region in the top portion of Figure 4.10. The remainder of the second wave of iterations can be initiated with the original stagger, and so on. Thus, the loop executes as $\lceil n/k \rceil$ waves of k iterations each, with successive iterations in a wave staggered by i, as depicted in Figure 4.10. Here we model two effects that cause the average parallelism to drop below $k\rho$: the fill-and-flush phenomenon observed in the unbounded case and the partial wave that occurs when k does not divide n. Observe, however, that the combined effect is fairly even. The worst case in terms of fill-and-flush is when k divides n, which is the best case in regard to the completeness of the final wave. The worst case in terms of the partial wave is when $n \bmod k = 1$, which is the best in regard to fill-and-flush. In between, moderate penalties are combined.

To gauge how reasonable the model is, we consider a simple example. Figure 4.11 shows profiles for Matrix Multiply with the outer loop bounded at 3 and the middle loop allowed

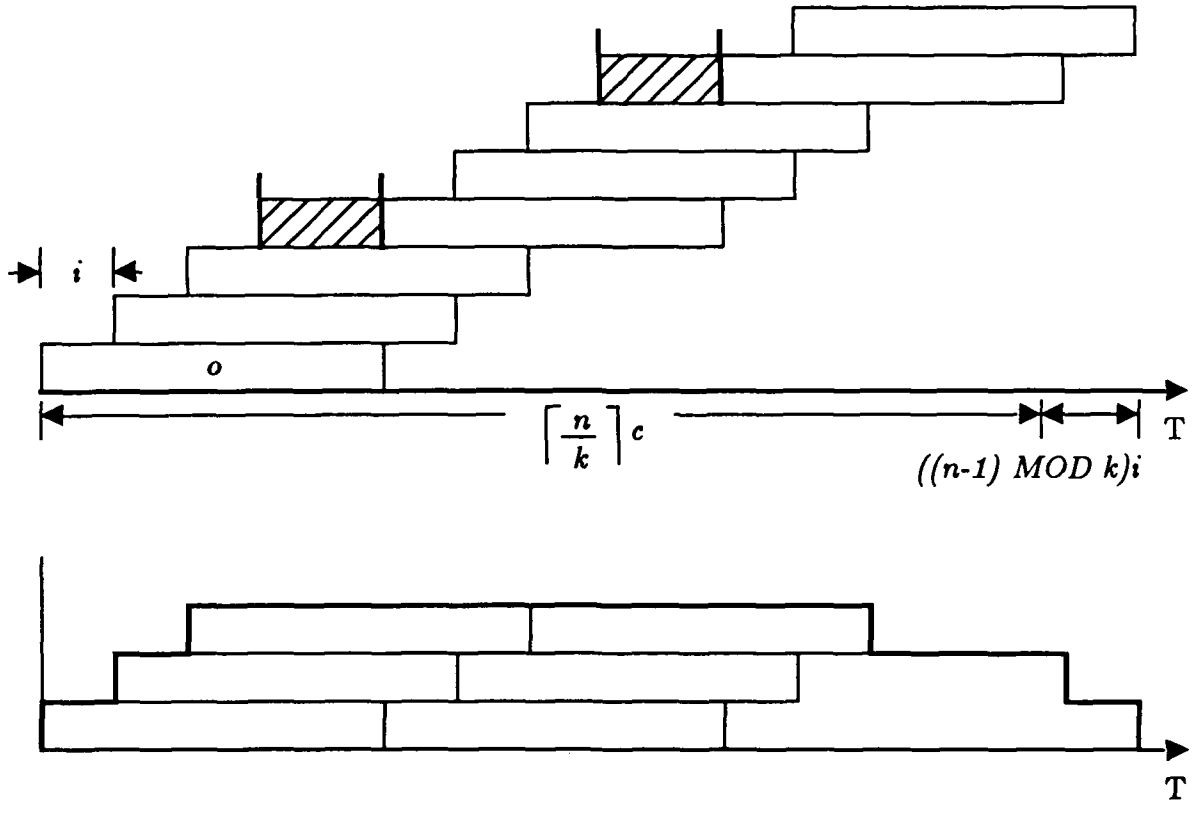


Figure 4.10: Model of k-Bounded Loop Unfolding (shown with $k = 3$)

to unfold completely It can be seen that execution does, indeed, proceed in six waves of three iterations, except the last which contains only one. However, the profile of a wave is the summation of the bell-shaped profiles of the three instances of the middle loop, since that is subject to fill-and-flush.

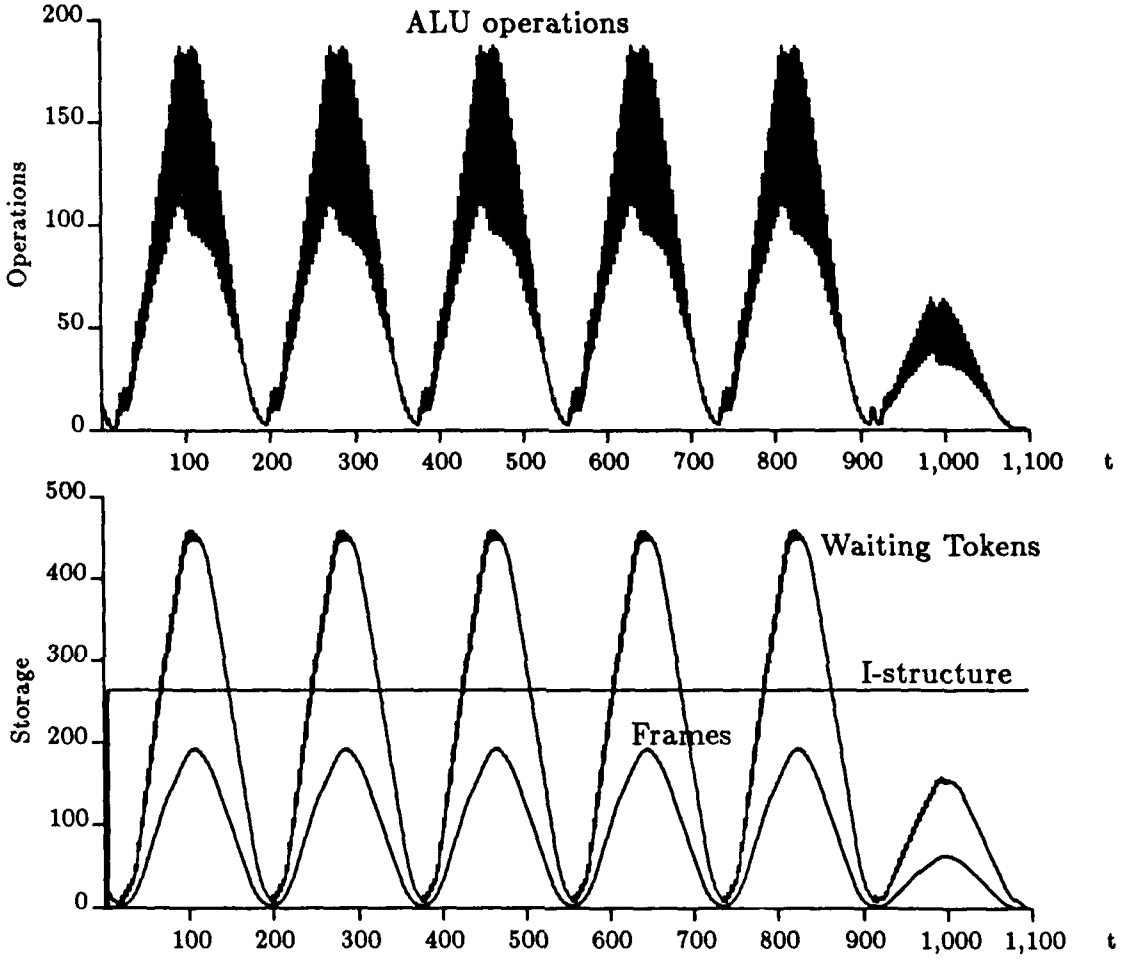


Figure 4.11: Ideal Parallelism and Resource Profiles for Matrix Multiply, 16, with Outer Loop Bounded at 3

Under our bounded loop model, the critical path of a k-bounded loop invocation can be given in terms of our structural parameters as

$$\tau_\infty(n,k) = \left\lceil \frac{n}{k} \right\rceil c + ((n-1) \bmod k)i. \tag{4.3}$$

Observe, that if i is small compared to c then there is almost no change in critical path for the range $n/2 < k < n$. Thus, as k is increased in this range, little increase in P_{ave} is obtained. Cytron observes a similar phenomenon[29] and suggests a local "trimming" optimization, which

can be stated in our terms as follows. Having arrived at loop bound k for a loop executing a total of n iterations, if k does not divide n it may be possible to reduce k without increasing the number of "waves" and, thereby, make more resources available elsewhere. We can obtain this by taking

$$k' = \left\lceil \frac{n}{\lceil \frac{n}{k} \rceil} \right\rceil .$$

The big pay-off here is when k falls in the "two-wave" range, discussed above.[7]

To validate our model of k-bounded loop behavior, we look once again at our Matrix Multiply example. Executing the program for all points in the space of k-bounds for the outer and middle loops, we get the data shown in Figure 4.6 above. The IP loop is kept at $k = 3$.

Applying (4.3) to the middle loop, we have

$$\tau_\infty(16, k_m) = \left\lceil \frac{16}{k_m} \right\rceil 110 + (15 \bmod k_m)4$$

and for the outer loop

$$\begin{aligned} \tau_\infty(16, k_o, k_m) &= \left\lceil \frac{16}{k_o} \right\rceil \tau_\infty(16, k_m) + (15 \bmod k_o)4, \\ P_{ave}(16, k_o, k_m) &= \frac{75,000}{\tau_\infty(16, k_o, k_m)} \end{aligned}$$

Figure 4.12 shows $\frac{P_{ave}(16,k_o,k_m)}{P_{ave}} \times 100$, the estimated parallelism as a percentage of the measured P_{ave}, over the entire k-space. $P_{ave}(16, k_o, k_m)$ is an underestimate everywhere, generally about 90% of the measured value, excluding the most tightly bound point.

4.4.4 Relationship of the model to graph structure

To ground this structural model, it is important to see in some detail how it relates to the structure of the dataflow graph itself. Simple loops with a single loop variable and no conditionals, such as the outer two loops of matrix multiply, can be cast into this model directly. The initiation rate, i, is simply the length of the index variable cycle, since the predicate can only be evaluated at the rate this variable circulates. The duration of an iteration is the length of the longest simple path in the graph from the output of the predicate, assuming no deferred reads, since each iteration initiates a traversal of this path.[8]

[7] Polychronopoulos presents a concept related to trimming, the *efficiency index of the loop*[77], which in our framework would be given by $\epsilon_i(k) = \frac{N_i/k_i}{\lceil N_i/k_i \rceil}$. This basically weights the partial wave effect by the number of waves.

[8] Deferred reads raise a variety of subtle issues, which we address briefly at the end of the thesis. In the main development we are interested in a crude characterization of the behavior of a loop, rather than a detailed

k_o															
16	92	93	94	93	95	95	97	97	97	97	96	96	96	95	100
15	89	89	91	90	91	90	92	91	91	91	91	90	90	89	94
14	91	90	91	89	91	90	92	92	91	91	91	91	90	89	95
13	91	90	91	90	91	90	92	92	91	91	91	91	90	90	95
12	91	90	91	90	91	90	92	92	92	91	91	90	91	89	95
11	91	90	92	90	91	91	92	92	92	91	91	91	91	90	95
10	91	90	90	90	91	90	92	92	92	91	91	91	90	89	95
9	89	90	92	91	92	91	92	92	92	92	91	91	91	89	95
8	89	90	92	90	92	91	93	92	92	92	91	91	91	90	95
7	90	88	90	88	89	88	91	90	90	90	90	89	89	88	93
6	90	90	89	90	89	89	92	90	90	90	90	89	89	87	93
5	91	87	89	89	88	90	89	88	88	89	89	88	88	86	91
4	87	90	91	89	89	89	91	91	90	90	89	89	88	87	91
3	93	90	90	90	89	87	90	90	88	89	89	87	86	86	90
2	83	93	86	87	89	90	89	87	90	87	88	86	88	84	90
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

k_m

Figure 4.12: $P_{ave}(16, k_o, k_m)$ as a Percentage of Measured $P_{ave}(k_o, k_m)$ Over Entire k-Space for 16×16 Matrix Multiply

The IP loop can be treated in the same fashion, even though it has two loop variables. This is because the cycle for **sum** is shorter than that for **k**, so iterations terminate at the rate they are initiated. The loop unfolds because values pipeline down paths from the **k** cycle to the **sum** cycle.

At this point, one might be led to believe that all loops have bounded unfolding under the fair scheduling embodied in our ideal execution model. Consider, however, the loop shown in Figure 4.13. The cycle involving the predicate is again of length four, so a new iteration is initiated every four steps. However, the cycle for the other loop variable is of length six, so the termination period is six steps. Thus, the number of concurrent iterations grows arbitrarily under fair scheduling. Tokens simply queue on the arc connecting the two cycles as the loop unfolds, as can be seen in Figure 4.14. However, even though this loop does not have a natural unfolding limit, it has a natural limit of *useful* unfolding. Suppose that the fast variable is slowed down to a period of six steps, after being allowed to get one iteration ahead. This does not change the critical path, since the control value is available at the *L-switch* when the data

prediction of its execution schedule. It is not difficult to embellish the model presented here to account for fixed communication latency in sending a value from a node in the graph to its successors; non-zero weights could be charged for such arcs. However, with deferred reads the latency can be arbitrarily large, even in the ideal execution model.

value of the slow cycle arrives, and furthermore where the paths merge in the body the data value is present by the time it can be used. By bounding the loop, the initiation rate is forced to match the termination rate in steady state, thereby slowing down the fast variable. Because this reduces the overhead in the clean-up of the allocate iterations, reducing the k-bound actually *improves* the critical path.

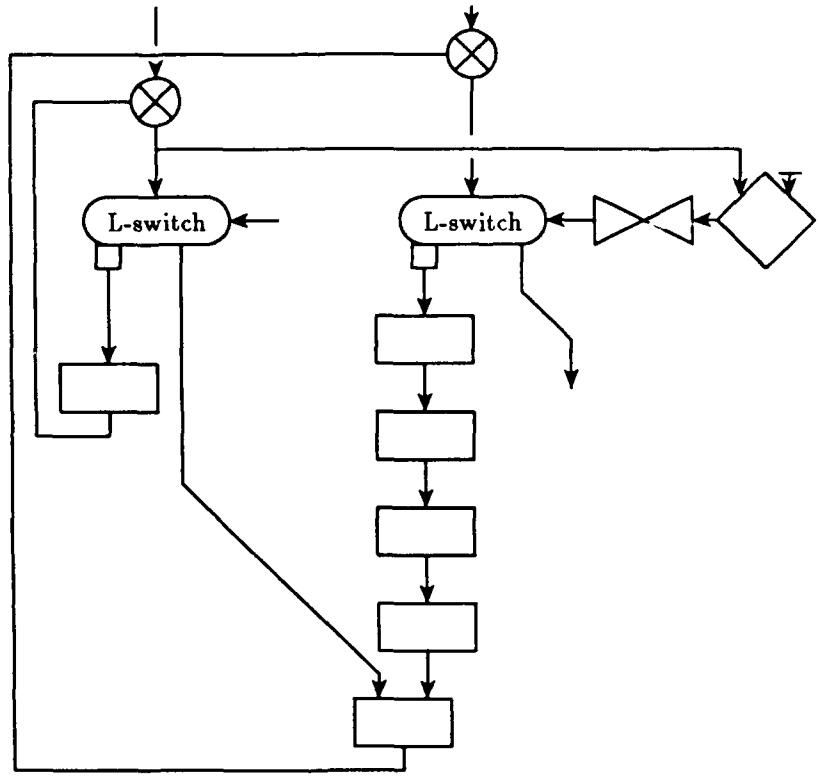


Figure 4.13: Example Loop with No Natural Unfolding Limit

In steady-state, the useful unfolding of a loop is determined by the longest cycle in the graph, whether or not it controls the predicate. The useful unfolding, μ, can be computed as the length of the longest path through the graph, divided by the length of the longest cycle, plus a small constant limited by the number of loop variables. Of course, in the context of conditionals, higher-order functions, and function calls, it may not be possible to determine the longest path, or even the true dependencies between loop variables. We will only be interested in applying this calculation directly in innermost loops, because elsewhere it is generally impractical to determine the three parameters. These innermost loops will often have a small μ, since c is small. The other important case, which is often difficult to detect, is where c is large, but $i \approx c$.

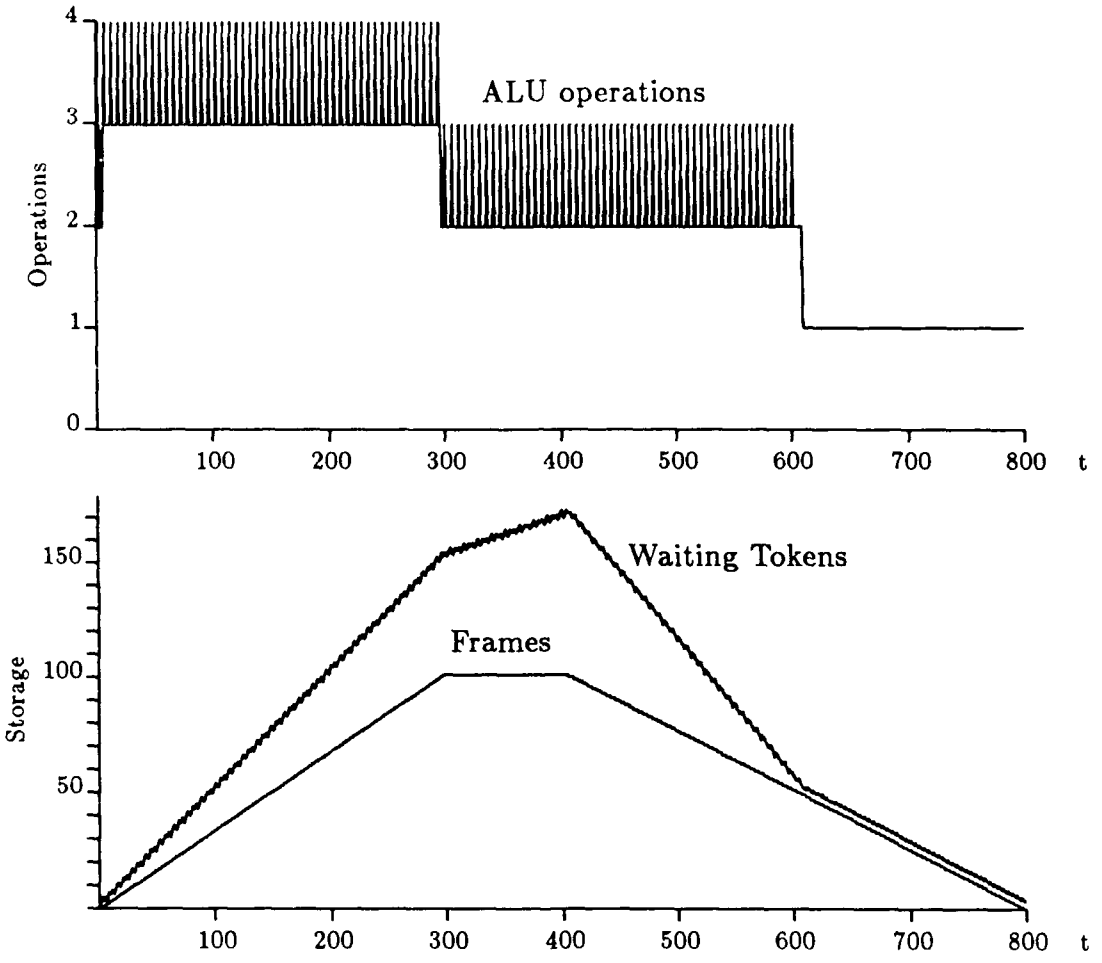


Figure 4.14: Ideal Parallelism and Resource Profiles for 100 Iterations of Example Loop with No Natural Unfolding Limit

4.4.5 Execution overhead

In supporting a k-bounded loop, the overhead, measured as the number of instructions executed, is proportional to k. There are a certain number of instructions executed to set up and clean up each allocated iteration. In the TTDA schema shown in the appendix, this overhead is $7k$, not including the cost of loop constants. The overhead in Monsoon graphs is considerably larger, about $30k$. Generally, loop constants will be stored in each activation frame or associated with the iterations in some manner. The empirical studies presented throughout the thesis involve a fixed cost per loop constant, independent of k, and thus have a certain artificial bias. In general, we will arrive at k-bound assignments that keep k for the innermost loops small, which implies that this bias also is fairly small. Recall, $P_{ave} \stackrel{\Delta}{=} \tau_1/\tau_\infty$, so it is the assumption that the total number of instructions, τ_1, is essentially constant that allows us to treat minimum critical path and maximum P_{ave} as identical goals.

4.5 Static Invocation Tree

In developing the method of estimating resources, we started with a single loop, then considered nesting, and then adjacent loops. Because the lexical structure and the dynamic structure of the programs we examined were closely related, we were able to draw informal loop structure diagrams and derive meaningful resource expressions. In this section, we introduce a static representation of the dynamic structure of loop programs, called the *static invocation tree*, which generalizes those diagrams. It provides the framework for building resource expressions and for characterizing the dynamic behavior of programs. Before defining the static invocation tree, let us review what we would like it to represent.

In Chapter 1, we presented the concept of an *execution tree*, describing the entire history of a program. Each code-block invocation and each iteration performed during execution of the program appears as a node in the tree and the children of a node represent the invocations it performs. Thus, a loop executing n iterations corresponds to n nodes, each of which may have several children, if the body includes several code-block invocations, or even a sequence of children, if it invokes a nested loop. Loops introduce a dynamic branching factor in the execution tree, while multiple invocations within a code-block introduce a static branching factor. For example, the execution tree for Matrix Multiply with $n = 4$, $m = 3$, and $s = 2$

would be as depicted in Figure 4.15.

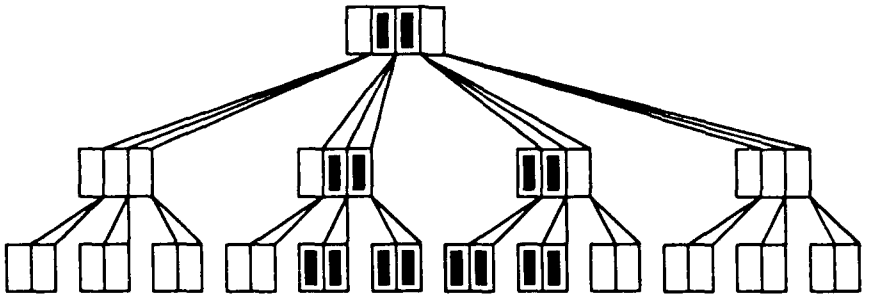


Figure 4.15: Execution Tree for Matrix Multiply, with (4×2) and (2×3) Matrices

At any point during the execution of a program, only a portion of the execution tree exists. Let us call this the *active subtree*. The size of the active subtree dictates the resource requirement of the program. Its size and shape dictate the parallelism offered. The k-bound for a loop limits the branch factor of the active subtree at the corresponding node. Thus, with all the loops bounded to two, the active subtree may be as depicted by darkened nodes in Figure 4.15. We are not concerned with exactly what part of the execution tree is active, but rather with the size and shape of the active subtree. This is precisely what is described by our resource expressions. By introducing a loop unfolding parameter at each loop node, we can form a concise summary of the active subtree.

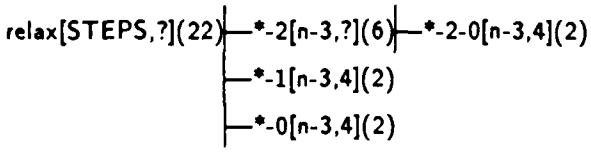
To characterize the resource and execution behavior of a program, we construct a static synopsis of the active portion of execution tree, called the *static invocation tree*. Starting at the top-level code-block, attach a child node for each code-block invocation appearing in the graph. The loop diagrams introduced informally above are a special case of this, where every code-block is a loop and is invoked at exactly one point in the program. More generally, a node may represent either an acyclic code-block or a loop, and in the latter case it is parameterized by the k-bound. The static invocation tree preserves the structure of the execution tree in a compact representation. A given setting of the k-bounds essentially determines the size and shape of the active subtree.

The restriction placed on the class of programs we consider in this thesis can now be formalized: we only consider those programs for which it is possible to construct the static invocation tree. The ability to do so is related to the more common compilation concept of a *call graph*, in which directed arcs represents the "calls" relation. In contrast, the static invocation tree

describes the dynamic context of each call. It can be constructed if the call graph contains no cycles. This proviso excludes general recursion, as well as use of higher-order functions beyond what can reasonably be analyzed statically. Experience with the current *Id* compiler shows that even in the presence of extensive and fairly sophisticated use of higher-order functions the static invocation tree can be formed, although this relies heavily on the use of tuple elision, arity analysis, and function in-lining to eliminate the "higher orderedness".

We will usually refer to the static invocation tree simply as the *call tree*. It provides a framework for program analysis and for recording the results of that analysis. Our notation for call trees is shown below for the Matrix Multiply program. Depth is indicated by levels increasing to the right. Internal loops are indicated by "*" followed by a sequence of numbers separated by dashes. The asterisk is short-hand for the lexically enclosing name and the numbers serve as unique identifiers. Square brackets indicate loop nodes and hold a pair of annotations. The first is a symbolic expression for the iteration count. The second is the maximum useful unfolding, μ, as determined by path length analysis. Where these cannot be derived by analysis a question-mark appears. The numbers in parentheses give information about the resource requirements of the active subtree rooted at the node under tightly constrained execution; these are discussed later.

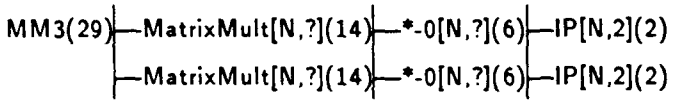
The policy for handling this structure will have to address nesting, as discussed earlier in this chapter. The call tree for **Relax** is shown below. It exhibits branching as well as nesting, plus we must deal with I-structure resources, not indicated here, in addition to local storage. The bounds employed in the different branches implicitly determine the ratio of resources available to the respective branches. So here we need guidelines for splitting resources across branches, as well as dealing with nesting.



Branching arises more generally through function composition. For example, a program that multiplies three square matrices could be expressed in *Id* as:

```
DEF MM3 A B C n s m = Matrixmult (MatrixMult A B n n n) C n n n);
```

and would generate the call tree shown below. Note that **MatrixMult** appears twice and the symbolic iteration counts are further specialized to reflect their common definition. Branching does not indicate, *a priori*, whether the two sub-computations will execute serially or in parallel.



A more realistic scenario is offered by the Simple code. The calling tree has nearly 300 nodes and extends seven levels deep. There are several overlapping phases of computation represented by the various branches and a variety of loops. However, some basic guidelines focus our attention rapidly on the crucial portion of the program.

Resource analysis is expressed in terms of the call tree, working from the leaves to the root. For a given setting of the k-bounds it is straightforward to estimate the size of the active subtree, and for the policies presented later it will be important to compute the size of each subtree assuming the k-bounds are at the minimum value. These numbers are shown for activation frames in parentheses in the call trees above. Symbolic resource expressions can be built by working up the tree as described above, however, we will not build these resource expressions explicitly for large portions of the tree. For a loop nest, we will build them implicitly in constructing the formulae that solve for the k-bounds. Through various kinds of dependence analysis, we identify substructures that are good candidates for unfolding, and loops that should not be permitted to unfold. The resource management code we add to the program essentially takes a resource parameter introduced at the root and "pushes" resources toward the leaves. This does not imply that the policy employed must have an "outer-in" form; in many cases we will be able to deal with several loops in a nest together.

4.6 Conclusions

We have given precise meaning to the term "loop bounding policy" and laid down how we will establish such a policy. This includes guidelines for analyzing the resource requirements of programs, structured in terms of the static invocation tree. Furthermore, we have developed

a simple model of the behavior of loops, in order to focus our efforts in analyzing potential parallelism.

The model developed here nicely predicts the behavior of very simple programs, and can serve as a basis for developing a loop bounding policy. One of the important concepts to emerge is that of a natural limit on the useful unfolding of a loop, μ. Already we have established the first component of our overall policy; loops that have small, fixed useful unfolding should be tightly bound. This can be determined for innermost loops by simple path-length analysis, but broader coverage can be obtained by more powerful analysis techniques, including interprocedural dependence analysis and array reference analysis[87, 88]. The model also offers some simple guidelines as to what sort of unfolding is most advantageous.

The next chapters deal with the various calling tree topologies as follows. Chapter 5 examines simple nested loops. Chapter 6 introduces the additional complexity of recurrences over data structures. Then, Chapter 7 examines branching. Finally, in Chapter 8 we carry out the resulting policy on a large program.

Chapter 5

Controlling Nested Loops

In this chapter and its sequels we study the behavior of progressively more complex program topologies and develop policies to deal with each, eventually covering the possible loop structures. Here we focus on nests of simple loops with no branching, except possibly in the innermost loop. We assume the recurrences in the outer loops are weak and the only resource of concern is local storage, *i.e.*, activation frames or tokens. I-structure storage is considered in the next chapter.

The goal here is to understand the effects of loop unfolding at various levels. We begin by exploring ramifications of the simple theory of loop behavior developed in the previous chapter. The observations arising from this study suggest a family of simple policies. We then evaluate these policies on a simple example, comparing the performance they achieve with the optimum achievable under the ideal execution model for several resource constraints. We also formulate and test two policies that are in opposition to the theoretical guidelines, to understand better the significance of the guidelines. Finally, as further sensitivity analysis, we test these policies on a small set of other programs with slightly different topologies.

5.1 Salient Properties of Loop Unfolding for Nested Loops

We begin our study of nested loops by examining implications of the simple theory of loop unfolding developed in Chapter 4. The construct we consider is a nest of `FOR` loops with loop bounds $k_1, k_2, \ldots, k_m$, where the only recurrence in the outer loops is the index variable and μ_i for each loop is fairly large. This generally implies that the inner loop contains some sizable computation.

The primary factor determining the amount of exposed parallelism is the number of concurrent instances of the computation in the inner loop, *e.g.*, the number of concurrent invocations of IP in our Matrix Multiply example. This number is simply the product of the k-bounds, so, for a fixed product of the k-bounds, we can consider the trade-offs in allowing unfolding at one loop level or another.

5.1.1 Parallelism

Observation 1 *For a fixed product of the k-bounds $\prod_{i=1}^{m} k_i$, the greatest parallelism is usually obtained when the unfolding of the outer loop is maximized.*

This is fundamentally a fill-and-flush argument; a single, long pipelined operation is better than a sequence of shorter pipelined operations because the fill-and-flush penalty is incurred once, not several times. The advantage of outer loop unfolding can be argued in terms of unfolding efficiency, but we take a more direct approach here.

Consider a doubly nested loop where the outer loop has bound k_1, executes a total of n_1 iterations, and has initiation rate i_1. Suppose the inner loop is similarly described by k_2,n_2, and i_2, with critical path per iteration c_2. Then, using (4.3), we have

$$\begin{aligned}
\tau_\infty(n_1,k_1,n_2,k_2) &= \left\lceil \frac{n_1}{k_1} \right\rceil \tau_\infty(n_2,k_2) + ((n_1-1) \bmod k_1) i_1 \\
&= \left\lceil \frac{n_1}{k_1} \right\rceil \left(\left\lceil \frac{n_2}{k_2} \right\rceil c_2 + ((n_2-1) \bmod k_2) i_2 \right) + ((n_1-1) \bmod k_1) i_1 \\
&= \left\lceil \frac{n_1}{k_1} \right\rceil \left\lceil \frac{n_2}{k_2} \right\rceil c_2 + \left\lceil \frac{n_1}{k_1} \right\rceil ((n_2-1) \bmod k_2) i_2 \qquad (5.1) \\
&\qquad + ((n_1-1) \bmod k_1) i_1
\end{aligned}$$

The first term in (5.1) is the "real work," while the second is the fill-and-flush penalty incurred in the inner loop and the last is the penalty incurred in the outer loop.

To simplify this, assume each k_i divides n_i, so we have

$$\tau_\infty(n_1,k_1,n_2,k_2) = \frac{n_1 n_2}{k_1 k_2} c_2 + \frac{n_1}{k_1}(k_2-1) i_2 + (k_1-1) i_1. \qquad (5.2)$$

If the loops are both simple FOR loops, then $i_1 = i_2$, and the penalty is minimized by maximizing k_1, for a given product $k = k_1 k_2$. The observation is stated in a fairly weak form, because non-linearities (due to the ceiling operations) introduce local perturbations.

This observation is substantiated by empirical measurements of our Matrix Multiply example, as shown in Figure 5.1. The top number in each entry is the measured P_{ave}. Entries on the upper-left to lower-right diagonals form contours of fixed k-bound product. In all cases, P_{ave} increases with k_o, the bound for the outer-loop.

k_o \ k_m	2	4	8	16
16	84 (**379**)	147 (669)	228 (1,230)	287 (2,187)
8	45 (195)	83 (**341**)	138 (629)	184 (1,190)
4	23 (103)	43 (177)	74 (321)	101 (609)
2	12 (56)	22 (93)	38 (165)	52 (309)

Figure 5.1: P_{ave} and (W_{max}) for 16 × 16 Matrix Multiply Under Various Loop Bounds

This observation suggests a policy favoring outer loop unfolding. However, (5.2) indicates that the advantage of outer loop unfolding is most pronounced at moderate resource levels. When resources are scarce, the product $k = k_1 k_2$ is small and both the penalty terms are small compared to the real work. When resources are plentiful, the product is large, say $k_1 k_2 \approx n_1^2$, and the two penalty terms are roughly equal. In the intermediate range, say $k_1 k_2 \approx n_1$, the effect is most pronounced.

5.1.2 Resource requirements

Observation 2 *For a fixed product of the k-bounds $\prod_{i=1}^{m} k_i$, the resource requirement is minimized by allowing unfolding of the inner loop.*

It is enough to look at the $m = 2$ case. The resource requirement of a (k_1, k_2) configuration is given by an expression of the form

$$R(k_1, k_2) = r_1 + k_1(r_2 + r_3 k_2). \tag{5.3}$$

For a fixed product $k_1 k_2$, this is minimized by minimizing k_1.

To validate this, the maximum token storage requirements for various k-bound configurations for Matrix Multiply are shown in parentheses in Figure 5.1. In all cases, these decrease

along the diagonals with increased unfolding of the middle loop, k_m. Fitting an equation of the form in (5.3) to the first few entries we get

$$W(k_o, k_m) = 9 + 5k_o + 9.5k_o k_m.$$

Note that in forming this estimate we are considering the maximum resource load at any point under the ideal execution model. The maximum load under any possible schedule may be higher. As noted previously, the token storage analysis technique of Appendix B gives a bound that is greater by a factor of three:

$$W_{est}(k_o, k_m) = 1 + 23k_o + 32k_o k_m.$$

This bound is somewhat loose, because it does not recognize the strictness of the innermost loop, and could be improved with more detailed analysis. The formula for activation frames is

$$A(k_o, k_m) = k_o + 4k_o k_m,$$

since the bound for the IP loop is fixed at 3. This is the worst-case under all execution orders and also fits the activation frame requirements measured under the ideal model very closely.

So far, we see that outer loop unfolding admits greater parallelism, but at greater resource cost, than inner loop unfolding. This is encouraging, since, from a policy viewpoint, the best situation is if all configurations offer the same parallelism per unit resource; then, they are all optimal. However, we need to examine the trade-offs in more detail to determine whether one effect dominates. Referring to Figure 5.1, observe that in going from configuration ($k_o = 16, k_m = 2$) to configuration ($k_o = 8, k_m = 4$) the reduction in P_{ave} is 1%, while the reduction in W_{max} is 11%. However, this savings does not continue for the next step along the diagonal. For local storage, the improvement in the resource requirement appears significant only for small values of the inner loop bound. Consider a pair of nested loops with resource expressions as given by (5.3). For a fixed product $k_1 k_2$, the absolute improvement due to doubling k_2 and halving k_1 is $r_2 k_1/2$. The relative improvement is no more than $\frac{r_2}{2r_2 + 2r_3 k_2}$. Assuming the r_i are roughly equal, for $k_2 > 4$ this falls below 10%.

More generally, for a fixed product of the loop bounds, $k = k_1 k_2$, we can determine how much the outer loop can unfold and keep the increase in resource usage, relative to the best case, under some fraction γ. After some algebraic manipulation, we get

$$k_1 \leq \gamma k \frac{r_3}{r_2}.$$

If we look at activation frame usage in our matrix multiply example, $r_3 = 3$ and $r_2 = 1$, so with $\gamma = 10\%$, we can let k_1 be as large as $k/2$. Thus, analysis of resource requirements suggests that a small amount of inner loop unfolding is valuable for most nested loops, but the return diminishes rapidly. Thus, a policy favoring outer loop unfolding can generally be improved by allowing a small amount of inner loop unfolding as well. A strong bias toward inner loop unfolding may have to be introduced when the resource coefficient for the outer loop is large, as we will see in the next chapter.

5.1.3 Execution overhead

Observation 3 *For a fixed product of the k-bounds $\prod_{i=1}^{m} k_i$, execution costs are minimized by allowing unfolding in the outer loop.*

> Using the basic k-bounded loop schema, the overhead for loop setup and cleanup in a k_o, k_m configuration is of the form $E(k_1, k_2) = e_1 k_1 + n_1 e_2 k_2$, for n_1 iterations of the outer loop. For all but trivial values of n_1, this is minimized by minimizing k_2. (If loop setup and cleanup are hoisted into the outer loop, the overhead is $E(k_1, k_2) = k_1(e_1 + e_2 k_2)$, which weakly favors inner loop unfolding.)

However, the real issue here is not the absolute overhead, but the overhead relative to the total work. For the TTDA schema used in our Matrix Multiply example, we have $e_i = 7$, so the overhead per unit k_m is 112, while the overhead per unit k_o is only 7 on a 16×16 problem. Setup and cleanup are not hoisted out of the middle loop. Even using the basic schema, the overhead due to setup and cleanup of the outer and middle loops amounts to no more than 2.5% of the execution cost, as shown in Figure 5.2. The bulk of the cost in this program is the IP loop, which is tightly constrained. In contrast, unrolling IP loop once reduces the overall execution cost by nearly 15%.

In the previous chapter, we observed that innermost loops have small useful unfolding ($\mu = c/i$), unless the loop body is quite large, *i.e.*, $c >> i$. When the loop body is large, the relative overhead due to k-bounding will be fairly small. On the other hand, when the loop body is small, the useful unfolding is small, so a large k-bound is useless, and the relative overhead per unit k is large. Therefore, it is very important to constrain small, deeply nested, innermost loops.

k_o		k_m			
	16	74,251	74,475	74,923	75,819
	8	74,195	74,419	74,867	75,763
	4	74,167	74,391	74,839	75,735
	2	74,153	74,377	74,825	75,721
		2	4	8	16

Figure 5.2: Dynamic Operation Count for 16×16 Matrix Multiply Under Various Loop Bounds

5.1.4 Partial-wave effect

Observation 4 *For a loop executing n_i iterations, $\frac{\partial P_{ave}}{\partial k_i} \approx 0$ for $n/2 < k_i < n$. Thus, k-bounds in this range make poor use of resources.*

This follows directly from (4.3), because in this range the reduction in critical path due to increasing the loop bound by Δk is only Δki. The loop executes as two waves, and increasing k simply depletes the second wave. Thus, if we were to compute P_{ave} over the entire space of k-bounds, we would expect levelling in the parallelism contour, as shown in Figure 5.3. The flat region for the outer loop is more serious than that for the inner loop, as can be seen in (5.1), but the intersection of the flat regions is invariably inefficient.

Examining Figure 4.6 of the previous chapter (page 68), we see that there is almost no increase in P_{ave} for $9 \leq k_o \leq 15$ at any k_m. This phenomenon is less pronounced for k_m in the same range. Figure 5.4 shows the resource cost in tokens per parallel operation throughout the k-space. Contours are drawn at eight tokens per operation.

The partial wave effect suggests that it is advantageous to avoid k-bounds in the range $n/2 < k < n$, regardless of which loop level is favored. However, the effect is most pronounced for outer loops. Furthermore, it is most important to avoid the upper end of this range.

5.2 Example Policies

In this section we examine a variety of simple policies that stem from the observations above and evaluate their performance on the Matrix Multiply example. We then explore two policies that do not reflect these observations to gauge further the importance of the theoretical guidelines. A policy defines a crude mapping of resource values to k-bound configurations, but the details

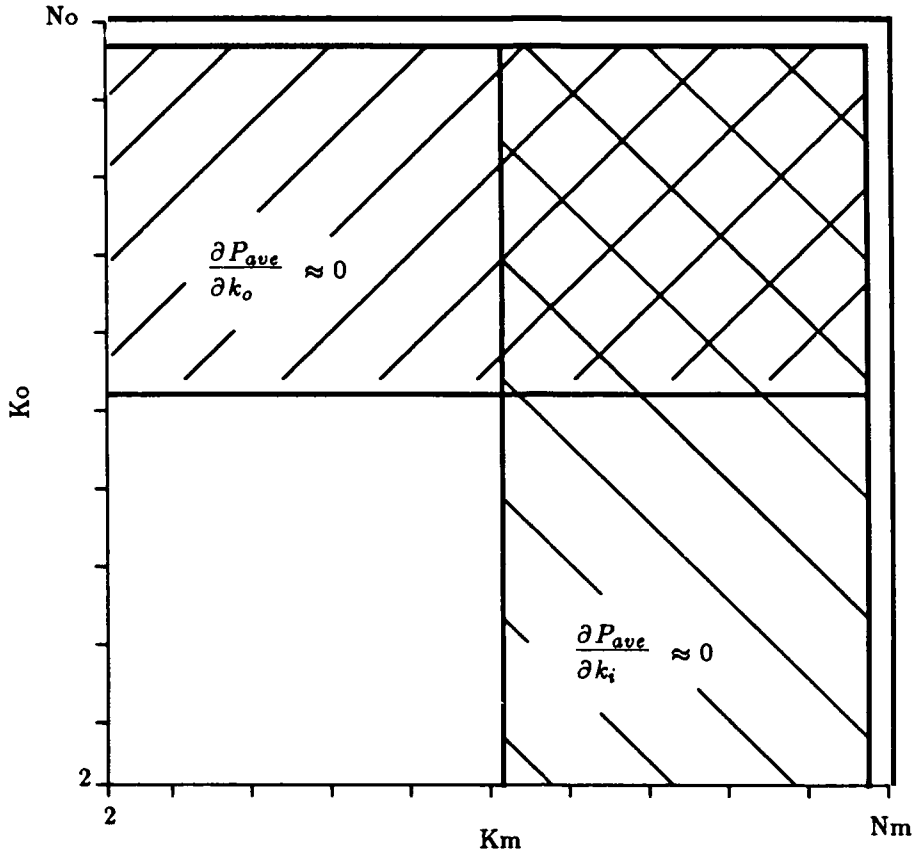


Figure 5.3: General Properties of k-Space

k_o	k_m = 2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	4.5	4.9	4.6	5.3	5.2	5.9	5.4	5.9	6.4	6.8	7.2	7.6	7.9	8.2	7.6
15	7.7	8.1	7.3	8.6	8.1	9.0	7.9	8.6	9.2	9.9	10.5	11.0	11.4	11.7	10.4
14	7.4	7.7	6.9	8.0	7.6	8.5	7.5	8.2	8.8	9.4	10.0	10.5	10.9	11.2	10.0
13	6.9	7.1	6.4	7.5	7.1	7.9	7.0	7.7	8.3	8.9	9.4	9.9	10.3	10.6	9.5
12	6.4	6.6	5.9	7.0	6.6	7.4	6.6	7.2	7.8	8.3	8.8	9.3	9.7	10.0	9.0
11	5.9	6.1	5.5	6.5	6.1	6.9	6.1	6.6	7.2	7.7	8.2	8.6	9.1	9.3	8.4
10	5.4	5.5	5.0	5.9	5.6	6.2	5.6	6.1	6.6	7.1	7.6	8.0	8.3	8.6	7.8
9	4.8	5.1	4.6	5.4	5.1	5.7	5.0	5.5	6.0	6.5	6.9	7.2	7.6	7.9	7.1
8	4.3	4.5	4.1	4.8	4.6	5.1	4.6	5.0	5.4	5.8	6.2	6.5	6.9	7.1	6.5
7	5.7	5.8	5.2	6.1	5.7	6.4	5.6	6.1	6.6	7.1	7.6	8.0	8.5	8.8	7.8
6	5.0	5.1	4.5	5.3	5.0	5.5	4.9	5.3	5.8	6.2	6.6	7.0	7.3	7.6	6.9
5	5.5	5.5	5.0	5.8	5.4	6.1	5.3	5.8	6.3	6.7	7.1	7.5	8.0	8.2	7.4
4	4.5	4.7	4.1	4.7	4.4	5.0	4.3	4.8	5.1	5.5	5.8	6.2	6.5	6.7	6.0
3	5.3	5.4	4.7	5.4	5.0	5.5	4.9	5.3	5.7	6.1	6.5	6.9	7.1	7.4	6.7
2	4.7	5.0	4.2	4.8	4.6	5.1	4.3	4.7	5.2	5.5	5.8	6.1	6.5	6.6	5.9

Figure 5.4: Tokens per Concurrent Operation (W_{max}/P_{ave}) Over Entire k-space for 16 × 16 Matrix Multiply

of the mapping are determined by the particular resource expression used. As much as the optimization viewpoint put forward in Chapter 4 may be an absurd method of program analysis, it provides a powerful empirical methodology for evaluating individual policies. We can compare the performance obtained with the k-bound assignment determined by a given policy for a resource constraint r against the best performance achievable within that constraint under the ideal model. This provides a common reference point for the various policies and indicates the overall room for improvement. An assignment may be suboptimal for two reasons: the policy may deviate from the optimal k-configuration on a given resource contour, or the resource expression may underestimate the feasible region.

We consider policies formulated against three resource expressions: activation frames, tokens as determined by the analysis in Appendix B, and tokens as determined empirically. As presented previously, for our Matrix Multiply example these are

$$\begin{aligned} A(k_o, k_m) &= k_o + 4k_o k_m \\ W_{est}(k_o, k_m) &= 1 + 23k_o + 32k_o k_m \\ W(k_o, k_m) &= 9 + 5k_o + 9.5k_o k_m \end{aligned}$$

The first two resource expressions are derived through program analysis, although the second overestimates waiting token usage under the ideal model. The last is derived by fitting a curve to resource values measured under the ideal model. The error under this approximation is shown in Figure 5.5 for a selection of points, indicating that the fit is quite good. It is, in fact, an upper bound everywhere. The A and W expressions characterize the feasible region under the ideal execution model accurately enough to isolate the effects due to choice of policy, while the W_{est} gives an indication of the cumulative effects.

For each resource expression, we produce a corresponding expression for each of the k parameters in terms of the others and the resource parameter. Given resource limit r and k-bound k_o, let $\widehat{K}_o(r, k_m)$ be the largest value of k_o such that $R(k_o, k_m) \leq r$ and similarly for $\widehat{K}_m(r, k_o)$. Thus, for frame storage limit A,

$$\begin{aligned} \widehat{K}_o(A, k_m) &= \max\left(\left\lfloor \frac{A}{1 + 4k_m} \right\rfloor, 2\right) \\ \widehat{K}_m(A, k_o) &= \max\left(\left\lfloor \frac{A - k_o}{4k_o} \right\rfloor, 2\right) \end{aligned}$$

k_o				
16	$\frac{385}{379} = 1.02$	$\frac{681}{669} = 1.02$	$\frac{1273}{1230} = 1.03$	$\frac{2457}{2187} = 1.12$
8	$\frac{197}{195} = 1.01$	$\frac{345}{341} = 1.01$	$\frac{641}{629} = 1.02$	$\frac{1233}{1190} = 1.04$
4	$\frac{103}{103} = 1.00$	$\frac{177}{177} = 1.00$	$\frac{325}{321} = 1.01$	$\frac{621}{609} = 1.02$
2	$\frac{56}{56} = 1.00$	$\frac{93}{93} = 1.00$	$\frac{167}{165} = 1.01$	$\frac{315}{309} = 1.02$
	2	4	8	16
		k_m		

Figure 5.5: $\frac{W_{expected}}{W_{max}} = \epsilon$ for Various Loop Bounds.

for token storage limit W,

$$\begin{aligned} \widehat{K}_o(W, k_m) &= \max\left(\left\lfloor \frac{T-9}{5+9.25k_m} \right\rfloor, 2\right) \\ \widehat{K}_m(W, k_o) &= \max\left(\left\lfloor \frac{T-9-5k_o}{9.25k_o} \right\rfloor, 2\right) \end{aligned}$$

and, for estimated token storage limit W'

$$\begin{aligned} \widehat{K}_o(W', k_m) &= \max\left(\left\lfloor \frac{W'-1}{23+32k_m} \right\rfloor, 2\right) \\ \widehat{K}_m(W', k_o) &= \max\left(\left\lfloor \frac{W'-1-23k_o}{32k_o} \right\rfloor, 2\right) \end{aligned}$$

These essentially reflect the code that is generated in augmenting the program to set the loop bounds.

5.2.1 Outer-In resource-limited policy

The first policy we consider derives from Observation 1. The outer loop is relaxed with the inner loop tightly bound until the resource limit or the outer loop iteration count is reached; after this point the inner loop k-bound is relaxed. This corresponds to a k-curve that traverses up the $k_m = 2$ line till $k_o = n$, and then along the $k_o = n$ line. To implement this, we compute k_o assuming k_m is at its minimum value, and then solve for k_m using this k_o, as follows:

$$\begin{aligned} k_o &= \min\left(\widehat{K}_o(r, 2), n\right) \\ k_m &= \min\left(\widehat{K}_m(r, k_o), n\right) \end{aligned}$$

The behavior of this policy for our Matrix Multiply example is shown in Figure 5.6 for several activation frame limits A. The first column gives the resource limit and the second and third show the k-configuration selected for that limit. The fourth column shows the peak

resource load with this configuration under the ideal execution model and the fifth shows this as a fraction of the resource limit. Note that in all cases the resource limit is honored, and at some point in the computation the resources are almost completely utilized. The sixth column shows the P_{ave} achieved under the selected k-configuration, the seventh shows the maximum P_{ave} achievable under the ideal model for this resource level, and the last column gives the ratio of these. The Outer-In policy falls short of the maximum achievable parallelism by 20% when resources are tightly constrained. Where the policy is suboptimal, the optimal k-configuration is indicated in the last column in parenthesis, (k_o, k_m). The results in the last column are summarized in Figure 5.9 by the points indicated with circles. The other points in that figure represent the behavior of the policies discussed below.

A	k_o	k_m	A_{max}	A_{max}/A	P_{ave}	Opt	$Pave/Opt$
50	5	2	45	0.90	23	28	(2,6) 0.82
100	11	2	99	0.99	45	56	(4,6) 0.80
200	16	2	144	0.72	84	106	(8,6) 0.79
400	16	6	400	1.00	183	183	1.00
600	16	9	592	0.99	232	232	1.00
800	16	12	784	0.98	242	242	1.00
1,000	16	15	976	0.98	254	254	1.00
1,200	16	16	1,040	0.87	287	287	1.00

Figure 5.6: Results of the Outer-In Frame-limited Policy on 16×16 Matrix Multiply for Several Activation Frame Limits

Figure 5.7 shows the behavior of this policy on our Matrix Multiply example under several token limits, W. In six out of ten cases it is optimal, and in all but one case, it is within 5%. It falls short when resources are tight and an improvement could be made by exchanging outer loop unfolding for inner loop unfolding. Excepting the case where the resource limit is below that of minimum unfolding, the resource limit is honored and resources are well utilized. These results are summarized in Figure 5.10 by the points indicated with circles.

Finally, Figure 5.8 shows the behavior of this policy when the formula derived from worst-case token storage analysis of the graph is employed. Here, we fall below optimal by a factor of three in most cases. These results are not surprising, since the analytic formula overestimates the actual requirements under the execution schedule implied by the ideal model by this factor. Below, we consider only the other two resource expressions in evaluating policies.

W	k_o	k_m	W_{max}	W_{max}/W	P_{ave}	Opt	$Pave/Opt$
50	2	2	56	1.12	12	12	1.00
100	3	2	80	0.80	15	22	(2,4) 0.68
200	8	2	195	0.98	45	45	1.00
400	16	2	379	0.95	84	84	1.00
600	16	3	524	0.87	108	109	(8,7) 0.99
800	16	4	669	0.84	147	147	1.00
1,200	16	7	1,094	0.91	187	187	1.00
1,500	16	9	1,366	0.91	232	235	(16,10) 0.99
2,000	16	12	1,742	0.87	242	249	(16,14) 0.97
2,500	16	16	2,187	0.87	287	287	1.00

Figure 5.7: Results of the Outer-In Token-limited Policy on 16×16 Matrix Multiply for Several Target Token Storage Limits

W	k_o	k_m	W_{max}	W_{max}/W	P_{ave}	Opt	$Pave/Opt$
50	2	2	56	1.12	12	12	1.00
100	2	2	56	0.56	12	22	0.55
200	2	2	56	0.28	12	45	0.27
400	4	2	103	0.26	23	84	0.27
600	6	2	149	0.25	30	109	0.28
800	9	2	218	0.27	45	147	0.31
1,200	13	2	310	0.26	45	187	0.24
1,500	16	2	379	0.25	84	235	0.36
2,000	16	3	524	0.26	108	249	0.43
2,500	16	4	669	0.27	147	287	0.51

Figure 5.8: Results of the Outer-In Token-limited Policy using Analytical Worst-Case Token Usage on 16×16 Matrix Multiply for Several Target Token Storage Limits

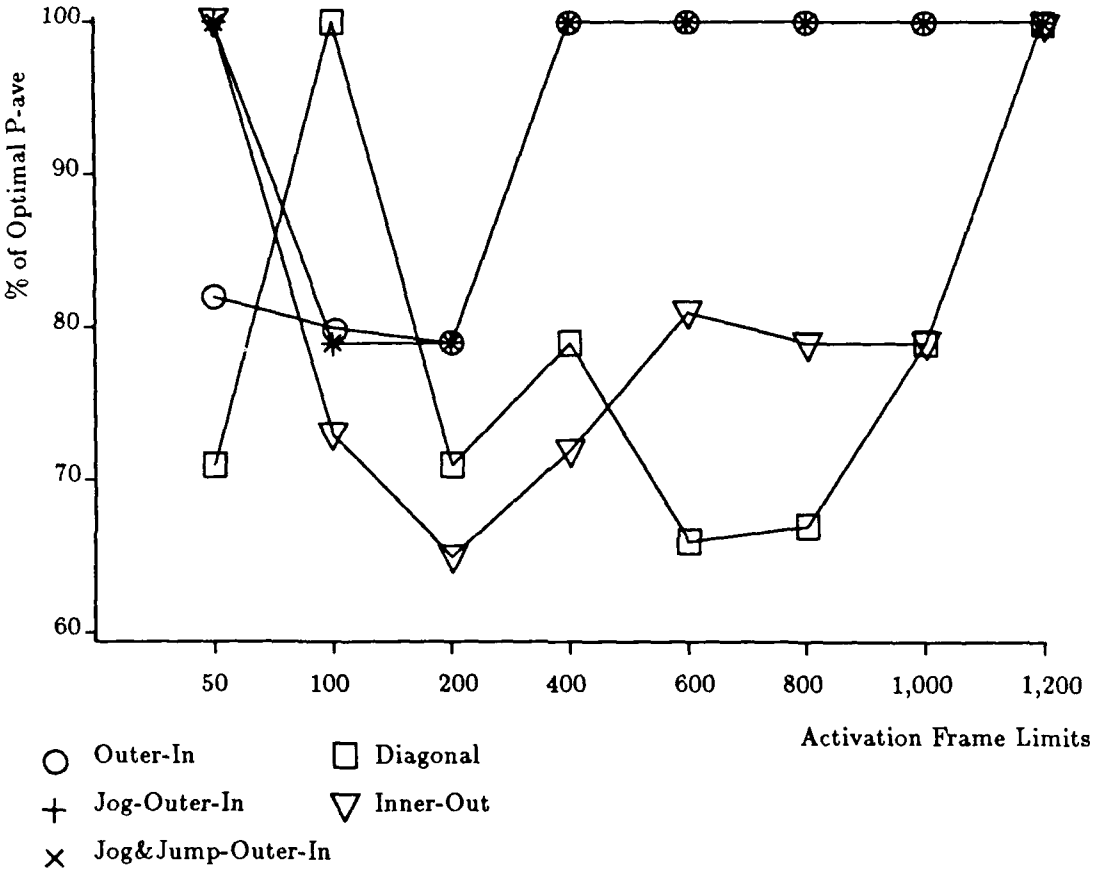


Figure 5.9: % Optimal P_{ave} for Five Activation Frame Limited Policies on Matrix Multiply, 16×16

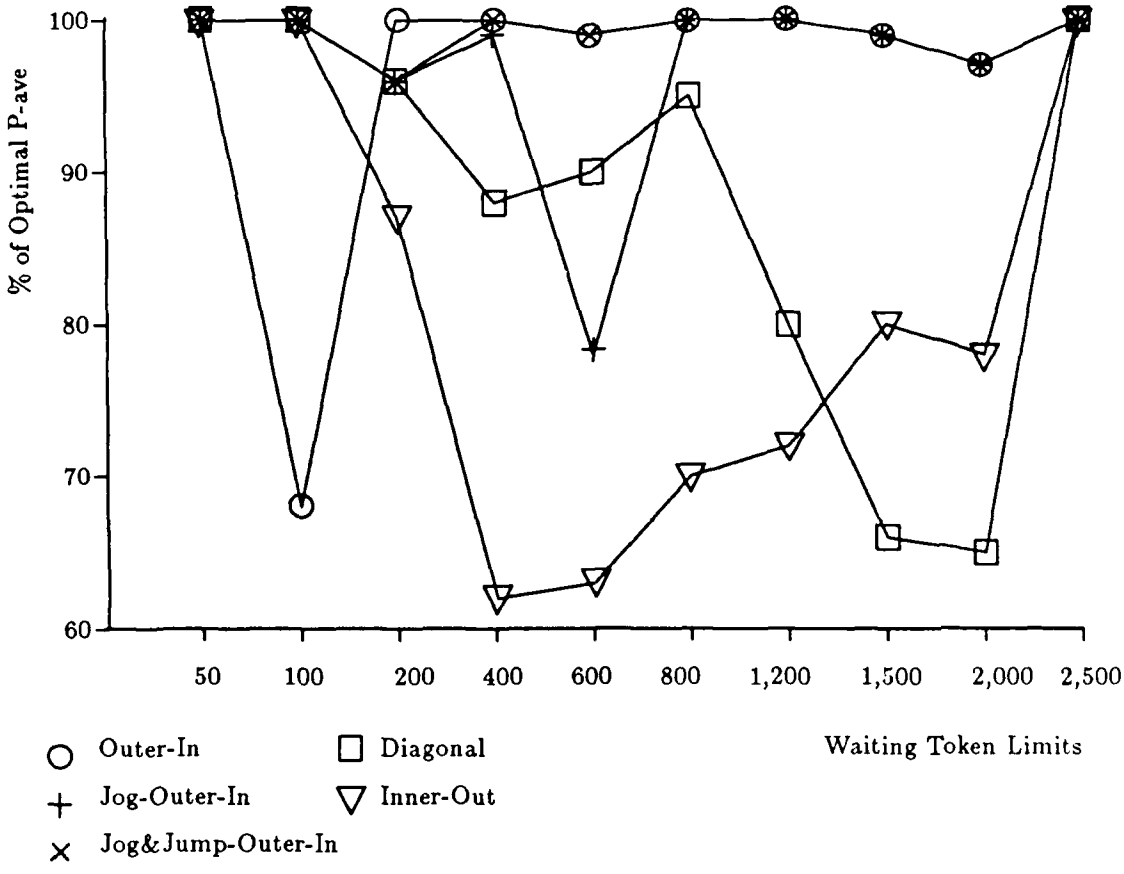


Figure 5.10: % Optimal P_{ave} for Five Waiting Token Limited Policies on Matrix Multiply, 16×16

5.2.2 Jog-Outer-In resource-limited policy

Observation 2 suggests that we should bias our policy to allow a small amount of inner loop unfolding, especially when resources are tight. To encompass this, we embellish the Outer-In policy to allow middle loop unfolding until $k_m = 4$, before allowing outer loop unfolding. To effect this, we make a first guess at k_m assuming $k_o = 2$, but setting an upper limit. Then we compute k_o based on this guess. Finally, we solve for k_m using k_o. Thus, our Jog-Outer-In Resource-Limited policy is given by:

$$\begin{aligned} k'_m &= \min\left(\widehat{K}_m(r,2),4\right) \quad \text{(Compute } k_m \text{ assuming } k_o = 2\text{)} \\ k_o &= \min\left(\widehat{K}_o(r,k'_m),n\right) \\ k_m &= \min\left(\widehat{K}_m(r,k_o),n\right) \end{aligned}$$

The behavior of this policy is shown in Figures 5.11 and 5.12 for activation frame and token storage limits, respectively. It is compared with the Outer-In policy in Figures 5.9 and 5.10, as indicated by plus signs (+). It performs better under tight resource limits than the simple Outer-In policy, but suffers at moderate resource limits, *e.g.*, $A = 200$ and $W = 600$. The improvement under tight resource limits is more pronounced for the token-limited policy, since the linear term in the resource expression is comparatively larger than that for activation frames. This means the resource contours are more asymmetric, favoring inner loop unfolding.

A	k_o	k_m	A_{max}	A_{max}/A	P_{ave}	Opt	$Pave/Opt$
50	2	6	50	1.00	28	28	1.00
100	5	4	85	0.85	44	56	(4,6) 0.79
200	11	4	187	0.94	84	106	(8.6) 0.79
400	16	6	400	1.00	183	183	1.00
600	16	9	592	0.99	232	232	1.00
800	16	12	784	0.98	242	242	1.00
1,000	16	15	976	0.98	254	254	1.00
1,200	16	16	1,040	0.87	287	287	1.00

Figure 5.11: Results of the Jog-Outer-In Frame-limited Policy on 16 × 16 Matrix Multiply for Several Activation Frame Limits

W	k_o	k_m	W_{max}	W_{max}/W	P_{ave}	Opt	$Pave/Opt$
50	2	2	56	1.12	12	12	1.00
100	2	4	93	0.93	22	22	1.00
200	4	4	177	0.88	43	45	(8,2) 0.96
400	9	4	382	0.95	83	84	(16,2) 0.99
600	14	4	587	0.98	85	109	(16,3) 0.78
800	16	4	669	0.84	147	147	1.00
1,200	16	7	1,094	0.91	187	187	1.00
1,500	16	9	1,366	0.91	232	235	(16,10) 0.99
2,000	16	12	1,742	0.87	242	249	(16,14) 0.97
2,500	16	16	2,187	0.87	287	287	1.00

Figure 5.12: Results of the Jog-Outer-In Token-limited Policy on 16 × 16 Matrix Multiply for Several Target Token Storage Limits

5.2.3 Jog&Jump-Outer-In resource-limited policy

The drop in performance of the Jog-Outer-In policy at moderate resource levels can be attributed to partial-wave effects, as predicted by Observation 4. At $W = 600$ the outer loop is in the two-wave regime. To avoid this, we embellish the Jog-Outer-In policy in a manner that forsakes the resource efficiency optimization to avoid situations where $n/2 < k_o < n$. If k_o falls in this range, we push it to $k_o = n$ and reduce k_m to compensate. This requires that the first guess at k_m is allowed to reach at least four before computing k_o, as for the Jog-Outer-In policy. This Jog&Jump-Outer-In Resource-Limited policy is given by:

$$\begin{aligned} k'_m &= \min\left(\widehat{K}_m(r,2),4\right) \\ k'_o &= \widehat{K}_o(r,k'_m) \\ k_o &= \text{if } k'_o > n/2 \text{ then } n \text{ else } k'_o \\ k_m &= \min\left(\widehat{K}_m(r,k_o),n\right) \end{aligned}$$

The behavior of this policy on our Matrix Multiply example is shown in Figures 5.13 and 5.14. The Token-Limited version is within 5% of optimal in all the test cases. The activation frame version shows no improvement, as the two cases where it is suboptimal are influenced by a bin-packing phenomenon — in decreasing k_o by small factor, a comparatively larger increase in k_m is possible. Behavior of the Jog&Jump-Outer-In policy is indicated by $\times$ in Figures 5.9 and 5.10.

A	k_o	k_m	A_{max}	A_{max}/A	P_{ave}	Opt	$Pave/Opt$
50	2	6	50	1.00	28	28	1.00
100	5	4	85	0.85	44	56	(4,6) 0.79
200	16	2	144	0.72	84	106	(4,6) 0.79
400	16	6	400	1.00	183	183	1.00
600	16	9	592	0.99	232	232	1.00
800	16	12	784	0.98	242	242	1.00
1,000	16	15	976	0.98	254	254	1.00
1,200	16	16	1,040	0.87	287	287	1.00

Figure 5.13: Results of the Jog&Jump-Outer-In Frame-limited Policy on 16×16 Matrix Multiply for Several Activation Frame Limits

W	k_o	k_m	W_{max}	W_{max}/W	P_{ave}	Opt	$Pave/Opt$
50	2	2	56	1.12	12	12	1.00
100	2	4	93	0.93	22	22	1.00
200	4	4	177	0.88	43	45	(8,2) 0.96
400	16	2	379	0.95	84	84	1.00
600	16	3	524	0.87	108	109	(8,7) 0.99
800	16	4	669	0.84	147	147	1.00
1,200	16	7	1,094	0.91	187	187	1.00
1,500	16	9	1,366	0.91	232	235	(16,10) 0.99
2,000	16	12	1,742	0.87	242	249	(16,14) 0.97
2,500	16	16	2,187	0.87	287	287	1.00

Figure 5.14: Results of the Jog&Jump-Outer-In Token-limited Policy on 16×16 Matrix Multiply for Several Token Storage Limits

5.2.4 Diagonal resource-limited policy

To gauge the quality of these policies, we also compare them with policies that run counter to our observations. Figures 5.15 and 5.16 show the behavior of a policy which relaxes the two k-bounds evenly. The performance of this policy is fairly good under tight resource limits, but falls substantially when the resource limit is loose, because the k-curve enters the plateau in the upper right quarter. The behavior of this policy is indicated by squares in Figures 5.9 and 5.10.

A	k_o	k_m	A_{max}	A_{max}/A	P_{ave}	Opt	$Pave/Opt$
50	3	3	39	0.78	20	28	0.71
100	4	6	100	1.00	56	56	1.00
200	7	6	175	0.87	75	106	0.71
400	9	10	369	0.92	145	183	0.79
600	12	12	588	0.98	154	232	0.66
800	14	14	798	1.00	162	242	0.67
1,000	15	16	975	0.98	200	254	0.79
1,200	16	16	1,040	0.87	287	287	1.00

Figure 5.15: Results of the Diagonal Frame-limited Policy on 16 × 16 Matrix Multiply for Several Activation Frame Limits

W	k_o	k_m	W_{max}	W_{max}/W	P_{ave}	Opt	$Pave/Opt$
50	2	2	56	1.12	12	12	1.00
100	2	4	93	0.93	22	22	1.00
200	4	4	177	0.88	43	45	0.96
400	6	6	367	0.92	74	84	0.88
600	7	8	552	0.92	98	109	0.90
800	9	8	706	0.88	140	147	0.95
1,200	11	11	1,155	0.96	150	187	0.80
1,500	12	12	1,354	0.90	154	235	0.66
2,000	14	14	1,768	0.88	162	249	0.65
2,500	16	16	2,187	0.87	287	287	1.00

Figure 5.16: Results of the Diagonal Token-limited Policy on 16 × 16 Matrix Multiply for Several Target Token Storage Limits

5.2.5 Inner-Out resource-limited policy

Finally, we consider a simple policy which favors the inner loop. This corresponds to a k-curve that starts going to the right and continues upward when the unfolding limit of the inner loop is reached. This is described by

$$\begin{aligned} k_m &= \min\left(\widehat{K}_m(r,2), n\right) \quad \text{(Compute } k_m \text{ assuming } k_o = 2\text{)} \\ k_o &= \min\left(\widehat{K}_o(r,k_m), n\right). \end{aligned}$$

As shown in Figures 5.17 and 5.18, it performs substantially worse than the family of Outer-In policies on our examples. This is clear in Figures 5.9 and 5.10, as indicated by triangles.

5.3 Variations

For nests of simple loops, a straightforward policy of relaxing loop bounds from the outermost inward performs quite well. Two optimizations on this policy show enough improvement to merit the increased complexity. On the other hand, all of the policies considered fall within a factor of two of optimal over the sample of test points. These results have been demonstrated for one particular example, but are consistent with the theory developed in Chapter 4 and should therefore be applicable to nests of simple loops in general. To complete this study, we need to understand how robust or fragile these policy guidelines are. In this section, we consider several variations from the simple nested loop scenario. First, we explore another simple nest where the inner loop has bounded unfolding. Then, we examine a case where there is a non-trivial recurrence in each of the loops, and yet significant potential unfolding in each. This example is extracted from the Simple code, discussed previously. Finally, we examine a variant of the problem studied in detail above, multiplication of lower triangular matrices, where the loop iteration limits are dynamic.

5.3.1 Another simple loop

We briefly examine a very simple loop nest, where the innermost loop has bounded, yet non-trivial, useful unfolding. The *Id* code for our simple relaxation example is shown below. This is similar to the program discussed in Section 2.3.2, but we have used conditionals to deal with the boundary conditions in order to remain within a strictly nested loop formulation. Path-length

A	k_o	k_m	A_{max}	A_{max}/A	P_{ave}	Opt	$Pave/Opt$
50	2	6	50	1.00	28	28	1.00
100	2	12	98	0.98	41	56	0.73
200	3	16	195	0.98	69	106	0.65
400	6	16	390	0.98	132	183	0.72
600	9	16	585	0.98	187	232	0.81
800	12	16	780	0.98	192	242	0.79
1,000	15	16	975	0.98	200	254	0.79
1,200	16	16	1,040	0.87	287	287	1.00

Figure 5.17: Results of the Inner-Out Frame-limited Policy on 16 × 16 Matrix Multiply for Several Activation Frame Limits

W	k_o	k_m	W_{max}	W_{max}/W	P_{ave}	Opt	$Pave/Opt$
50	2	2	56	1.12	12	12	1.00
100	2	4	93	0.93	22	22	1.00
200	2	9	183	0.92	39	45	0.87
400	2	16	309	0.77	52	84	0.62
600	3	16	459	0.76	69	109	0.63
800	5	16	758	0.95	103	147	0.70
1,200	7	16	1,049	0.87	134	187	0.72
1,500	9	16	1,331	0.89	187	235	0.80
2,000	13	16	1,846	0.92	194	249	0.78
2,500	16	16	2,187	0.87	287	287	1.00

Figure 5.18: Results of the Inner-Out Token-limited Policy on 16 × 16 Matrix Multiply for Several Target Token Storage Limits

analysis gives $\mu = 5.0$ for the inner loop. Figure 5.19 shows P_{ave} for this example on a 10×10 mesh for the entire k-space. Indeed, there is no increase in parallelism for inner loop unfolding beyond this point ($k_i = 5$). There is little increase in parallelism beyond $k_i = 3$.

```
defsubst relax_step' A =
  {((l1,u1),(l2,u2)) = (2D_bounds A);
   R = I_matrix (2D_bounds A);
   {for i <- l1 to u1 do
      {for j <- l2 to u2 do
         il = if i == l1 then i+1 else i-1;
         ih = if i == u1 then i-1 else i+1;
         jl = if j == l2 then j+1 else j-1;
         jh = if j == u2 then j-1 else j+1;
 R[i,j] = stencil A il i ih jl j jh}}
   in R};
```

k_o										
	10	[36]	[42]	[46]	[49]	49	48	47	47	46
	9	24	31	35	39	38	37	36	35	34
	8	24	30	34	37	37	36	35	34	33
	7	23	29	33	36	36	35	34	33	32
	6	23	[29]	33	36	35	34	33	33	32
	5	[23]	[28]	[32]	[35]	34	34	33	32	31
	4	16	21	[24]	26	26	25	24	24	23
	3	13	16	[19]	21	21	20	19	19	18
	2	[10]	[13]	[15]	[17]	16	16	15	15	14
		2	3	4	5	6	7	8	9	10
					k_i					

Figure 5.19: P_{ave} over k-Space for Relax_step', 10×10, with A_{max} Optimal Points Marked

5.3.2 Non-trivial recurrences

To examine the validity of our nested loop policy guidelines in the context of non-trivial recurrences in each loop, we consider a loop nest extracted from the Simple code. At each point in the mesh a fairly complex expression is evaluated to determine the maximum time-step that preserves the Courant condition, *i.e.*, keeps the fluid from moving across too many zones. This is minimized over the entire mesh by computing the minimum of each row as a linear recurrence in the inner loop and the minimum of these as a linear recurrence in the outer loop. The calling tree for this problem, shown below, has a doubly nested loop, with calls to a square root function in the inner loop.

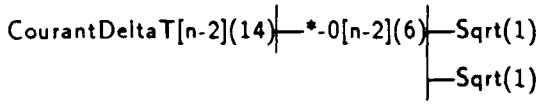


Figure 5.20 shows the measured P_{ave} over the entire k-space on a 10×10 mesh. As observed in simpler nests, outer loop unfolding exposes more parallelism than inner loop unfolding. The maximum activation frame (A_{max}) and token (W_{max}) requirements measured under the ideal model are shown in Figures 5.21 and 5.22, respectively. We see that the resource requirements favor inner loop unfolding, as observed in the above examples, and the asymmetry in resource contours is less than that for P_{ave}.

The crucial information from a policy viewpoint is the nature of the optimal k-curve; this is indicated by boxed entries in Figure 5.20. It strongly suggests outer loop unfolding, with a slight amount of inner loop unfolding under tight resource constraints, again consistent with the observations above. Finally, the behavior of the various policies developed above is summarized in Figure 5.23. All of the Outer-In policies perform quite well, with the other two policies falling short in a fashion similar to that in our Matrix Multiply example.

k_o \ k_m	2	3	4	5	6	7	8	9	10
10	44	60	63	74	76	78	82	87	88
9	44	60	63	74	76	78	82	87	88
8	26	37	40	49	51	53	56	61	62
7	26	37	39	48	50	52	55	59	60
6	25	36	39	48	49	51	54	58	59
5	25	36	38	47	48	50	53	57	58
4	17	25	27	34	35	36	38	42	42
3	17	25	27	33	34	35	37	40	41
2	11	15	17	21	21	22	23	26	26

Figure 5.20: P_{ave} for CourantDeltaT on 10×10 Mesh over k-Space with A_{max} Optimal Points Marked

k_o \ k_m	2	3	4	5	6	7	8	9	10
10	61	86	110	134	158	181	199	212	217
9	61	86	110	134	158	181	199	212	217
8	55	78	100	121	142	163	180	193	198
7	48	69	89	108	126	144	161	173	177
6	41	59	77	94	110	125	140	151	155
5	34	49	64	79	93	106	118	129	132
4	27	39	51	63	75	86	96	105	107
3	20	29	38	47	56	65	73	80	82
2	13	19	25	31	37	43	49	54	55

Figure 5.21: A_{max} for CourantDeltaT on 10×10 Mesh Over k-Space

k_o \ k_m	2	3	4	5	6	7	8	9	10
10	300	404	508	599	688	779	856	905	915
9	297	401	505	596	685	776	853	902	912
8	275	378	468	557	633	708	784	832	842
7	252	341	430	505	580	641	701	748	757
6	221	304	379	453	513	573	619	664	672
5	185	258	327	387	446	491	536	568	575
4	150	208	266	320	365	409	440	470	476
3	117	159	201	244	283	313	343	359	364
2	80	109	136	164	193	217	233	248	250

Figure 5.22: W_{max} for CourantDeltaT on 10×10 Mesh Over k-Space

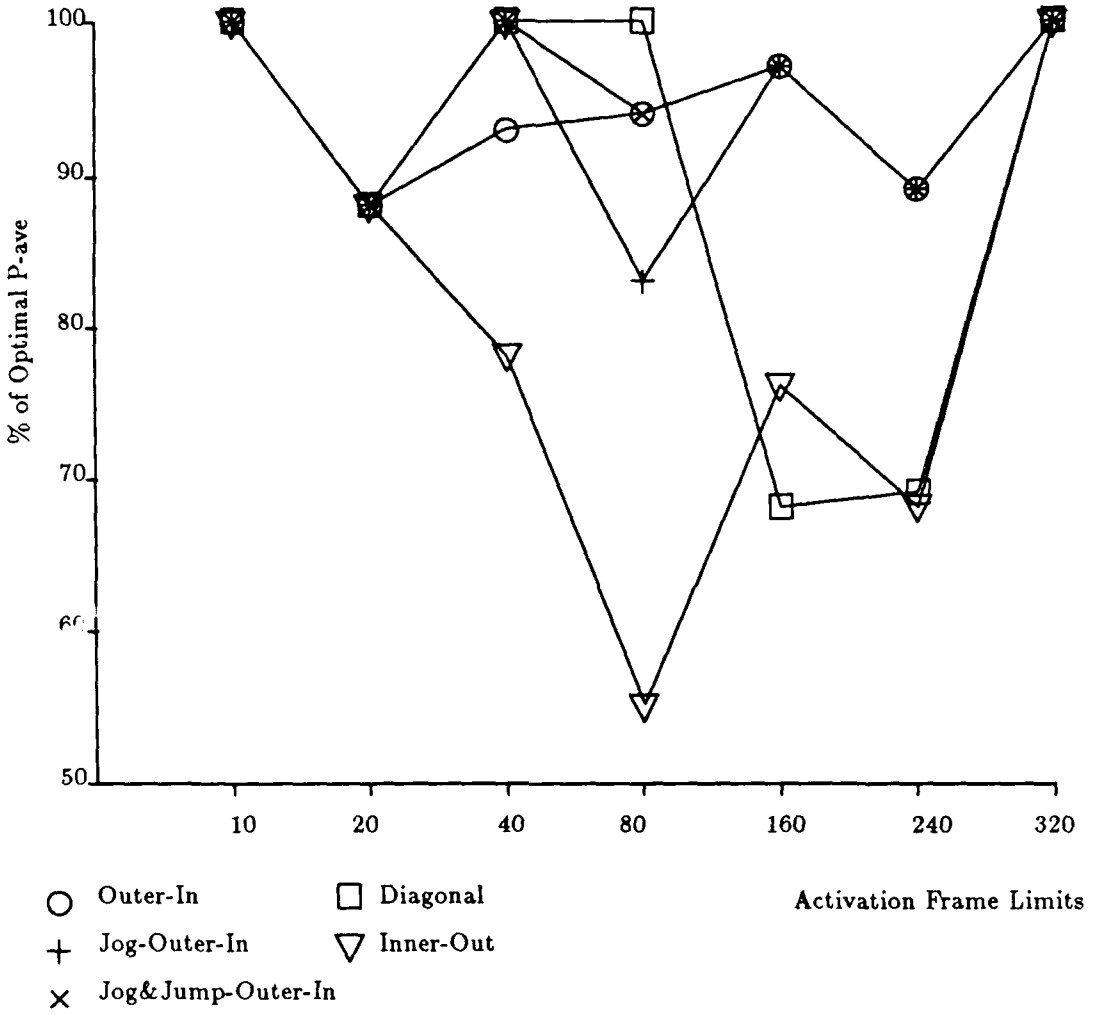


Figure 5.23: % Optimal P_{ave} for Five Activation Frame Limited Policies on CourantDeltaT, 10×10

5.3.3 Dynamic iteration counts

The last variation we explore is one in which the number of iterations is not a simple function of top level parameters. An extreme case is when the iteration count cannot be determined even at the time the loop is invoked, as with WHILE loops. Often when such situations arise, there is also a strong recurrence and the loop has bounded useful unfolding, although detecting this can be difficult. The Simple code provides two interesting examples of this ilk. There is an inner loop that performs a table look-up. Path-length analysis gives a very tight bound, so the complexity of the iteration structure is unimportant. However, this loop is invoked within a loop that inverts a function of two variables; here, fairly sophisticated dependence analysis is required to show that, in fact, the outer WHILE loop does not unfold either. There are other important classes of loops that exhibit substantial unfolding, yet we cannot determine the iteration count upon loop invocation. List traversal, which is common in sparse matrix routines, is a prime example. The Outer-In policies we have discussed break down if such a loop appears as the outer loop, since we cannot determine if the inner loop should unfold. Inner-Out policies break down in an analogous fashion. The simplest way to avoid these situations is to try and avoid unfolding loops of this form when possible, *i.e.*, if there is a way of exposing enough parallelism at reasonable resource requirements without letting such loops unfold, follow that policy. If a degree of freedom remains after recognizing as many constraints as possible, follow a middle path.

A simpler situation that arises quite frequently is where the iteration count is determined by values determined within the program, such as the elimination step of LUP decomposition on smaller and smaller submatrices. The flexibility of k-bounds handles this case nicely; it only means that the k-bound determination cannot be lifted out of the loop. As an example, we consider a somewhat contrived program that multiplies lower triangular matrices. The *Id* code is shown below, augmented with determination of loop bounds in terms of a top-level resource parameter **Frames**. Ideal parallelism and resource profiles for this are shown in Figure 5.24 for 50 frames. The behavior is quite good until the iteration counts get too small to generate significant parallel activity.

```
DEF LIP A B i j = {sum = 0
                   in {FOR k <- j to i bound 2 do
                         next sum = sum + A[i,k]*B[k,j];
                       FINALLY sum}};
DEF LMMr A B n Frames =
  {C = I_matrix ((1,n),(1,n)) ;
   k1 = (max 2 (min n (floor (Frames / 7))));      % compute k1 assuming k2 = 2
   {for i <- 1 to n bound k1 do
      k2 = (max 2 (min i (Frames - k1) / (3*k1))); % compute k2 given k1
      {for j <- 1 to i bound k2 do
         C[i,j] = lip A B  i j}}
   in C};
```

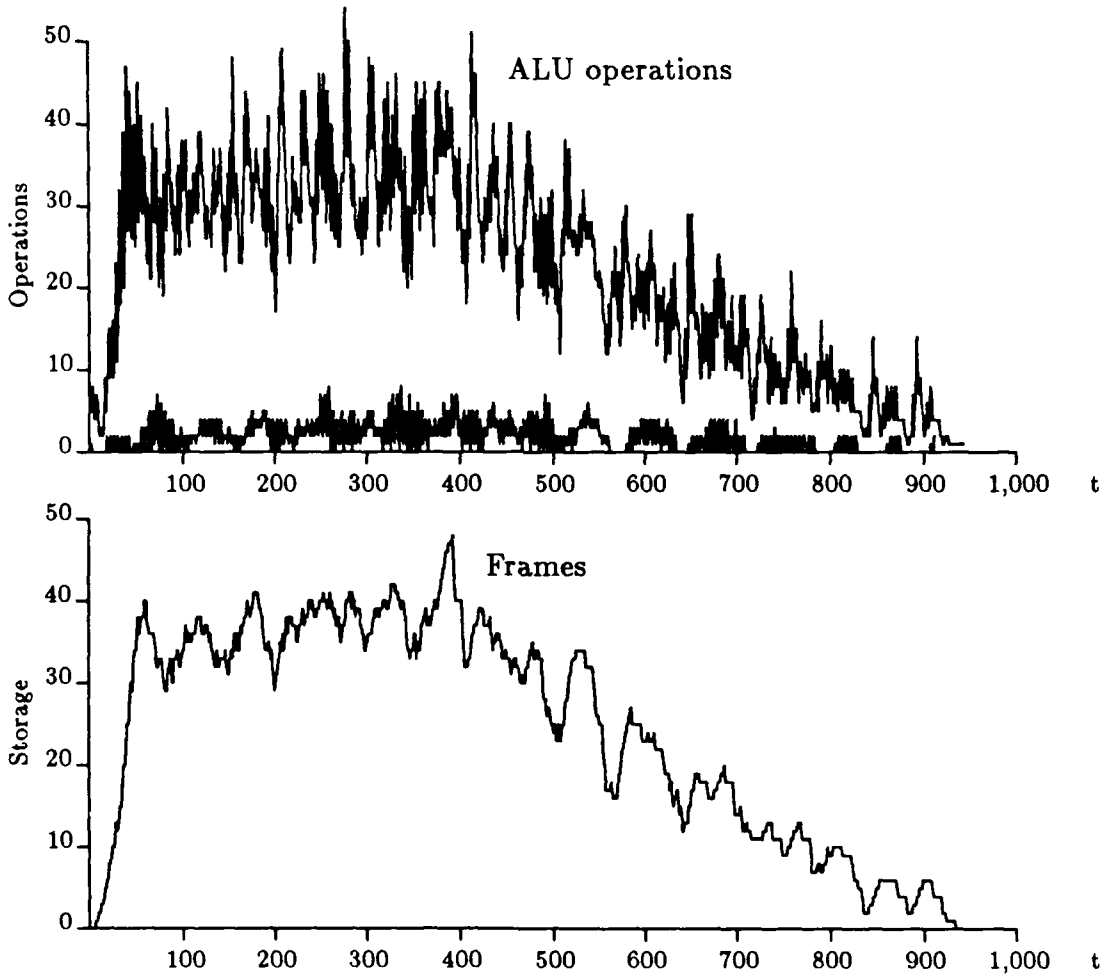


Figure 5.24: Ideal Parallelism and Resource Profiles for LMMr, $n = 16$ and $Frames = 50$

5.4 Conclusions

For simple nests of loops, in the absence of structure allocation or strong scalar recurrences in the outer loops, simple policies favoring outer loop unfolding perform quite well. The policy of choice is primarily Outer-In, with the "Jog&Jump" optimization introducing a small amount of inner loop unfolding when resource constraints are tight. Thus, our Matrix Multiply program should be augmented with resource management code to assign loop bounds as shown underlined below.

```
DEF MatrixMult' A B n s m frames =
  {k1 = floor ((frames - 1) / 17);
   ko = if (k1 > n/2) then n else (max 2 k1);
   km = max 2 (((frames/ko) - 1) / 4);
   in  {matrix ((1,n),(1,m)) | [i,j] = ip A B s i j
        || i <- 1 to n bound ko
& j <- 1 to m bound km}};
```

The quality of this policy has been demonstrated on a few small examples and is consistent with our model of loop behavior. It represents a compromise between fill-and-flush overhead, resources, and bin-packing anomalies. In the examples studied here the initiation rates for the two loop levels are equal, although in CourantDeltaT both the index variable and the minimum value are circulated. Expression (5.2) gives a measure for how strong the outer loop dependence (i_1) would have to be relative to that of the inner loop (i_2) before the fill-and-flush penalty of the outer loop would dominate. Although the cross-over point can be worked out in detail, the short answer is "very strong." Generally, where there is such a strong dependence from one iteration to the next, there is little useful unfolding, as well. The role of detailed dependence-analysis in this paradigm is determining where the simple Outer-In policies break down.

Chapter 6

Controlling Loops with Structure Recurrences

In this chapter we extend our study of nested loops to deal with the interaction of two resource constraints: token storage and I-structure storage. We look at a nest of loops where a data structure is allocated in each iteration of the outer loop and used by a small window of iterations. The most common construct of this form is where the outer loop is a linear recurrence over a large data structure. More traditional methods of exposing parallelism would not attempt to uncover the parallelism available in a loop of this form, but dynamic dataflow does so automatically. Under resource-limited execution, this gives rise to interesting trade-offs. For example, it is possible to constrain the computation for each version of the data structure and allow more versions, or to relax the computation on each version and restrict the number of versions. We explore two situations, one in which the combined token and structure requirement is limited and one in which the two resources have independent constraints. In both cases, we find that inner loop unfolding is clearly superior to outer loop unfolding, in stark contrast to the class of loop nests studied in the previous chapter. This leads to a simple classification scheme for loops: those with limited useful unfolding, those that have expensive unfolding, and, finally, those that are good candidates for unfolding.

6.1 Salient Features

In re-examining the salient features of loop nests in this new context, we see that the observations of the previous chapter that motivated a class of Outer-In policies no longer hold. With

simple loop nests we found that maximizing parallelism and minimizing resources pulled in opposite directions, so all intermediate choices were fairly good and the best choice was determined by the strongest pull in a given region. In this new context, we find that both concerns pull strongly in the direction of inner loop unfolding.

6.1.1 Parallelism

A loop that allocates large arrays of known lifetime in each iteration will usually involve a recurrence on the arrays. In dealing with simple loops, we operated under the assumption that loops other than the innermost were in the useful unfolding regime, *i.e.*, $n < \mu$. However, the presence of the recurrence in the outer loop may violate this assumption. Hence, the fill-and-flush argument of the previous chapter may not hold. Although the loop may unfold completely, allocating many versions of the data structure, it may have limited *useful* unfolding. Determining the presence of this outer loop recurrence is not too difficult in most cases, although translating the strength of this recurrence into a limit on natural unfolding, μ, is difficult. Simple path-length analysis of the loop body is not sufficient. We would need to determine the length of the element-wise dependencies from one version of the structure to the next and compare that with the rate that the inner loops sweep through a *single version of the structure*. Fortunately, resource limitations will usually place a fairly tight constraint on such loops, so our inability to perform this analysis is not too serious.

6.1.2 Resource requirements

The allocation of a structure in each per outer loop iteration changes the resource behavior of these loops dramatically. There are now two resources placing constraints on loop unfolding. Suppose we have local storage capacity T and structure storage capacity S, *i.e.*, we are working with an architecture with distinct memory spaces for the two forms of storage. For a nested pair of loops, we need to optimize for k_o, k_i under the constraint

$$\begin{aligned} T(k_o, k_i) &\leq T \text{ and} \\ S(k_o, k_i) &\leq S, \end{aligned}$$

where the resource expressions $T(k_o, k_i)$ and $S(k_o, k_i)$ are given by

$$T(k_o, k_i) = t_1 + t_2 k_o + t_3 k_o k_i \text{ and}$$

$$S(k_o, k_i) \quad = \quad s_1 + s_2 k_o.$$

Like the loops studied in the previous chapter, the local storage contours (token or activation frame) are hyperbolas, as shown in Figure 6.1. They are slightly asymmetric, weakly favoring inner loop unfolding for a given product $k_o k_i$. The structure storage expression is independent of k_i, so the corresponding contours are horizontal. Thus, structure constraint S serves to place an upper limit on k_o, but places no constraint on k_i.

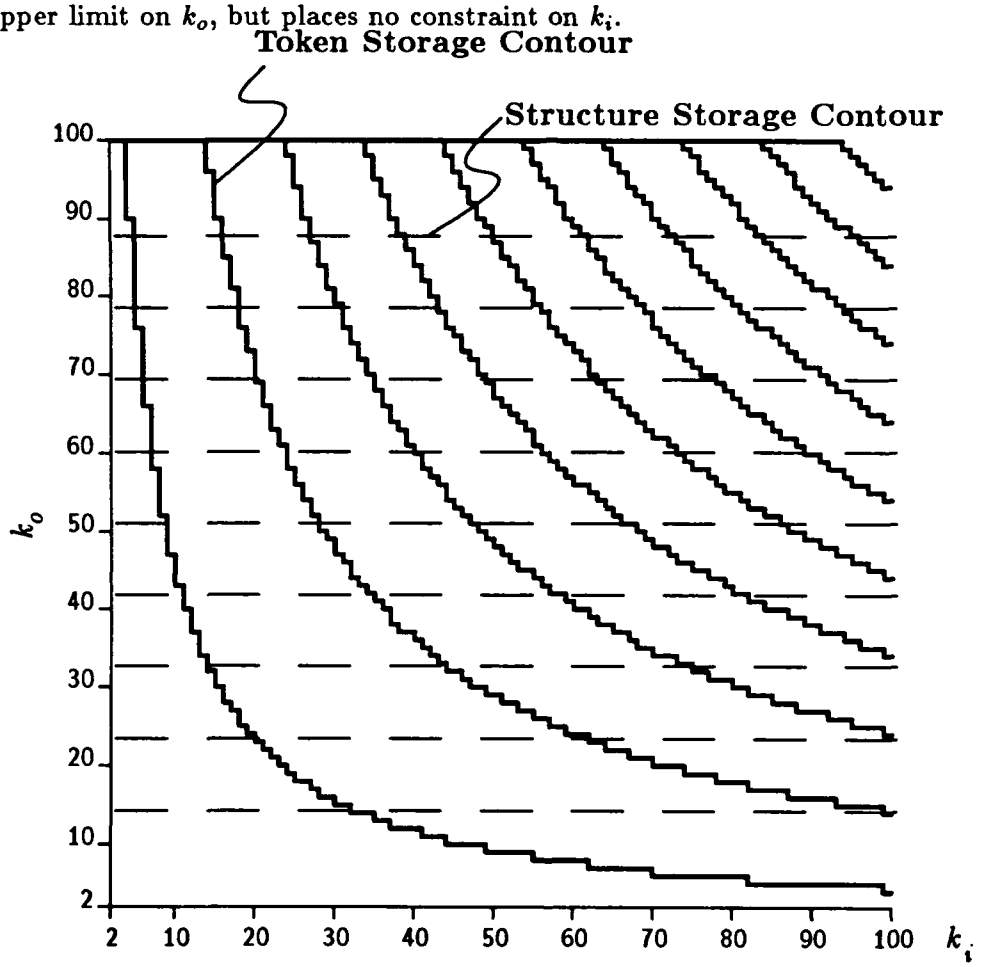


Figure 6.1: Typical Independent Token and Structure Storage Contours

The alternative approach, which may be attractive for machines with a single storage pool, is to consider a single constraint r on the combined storage usage:

$$R(k_o, k_i) \quad \leq \quad r, \text{ where}$$

$$R(k_o, k_i) \quad = \quad (t_1 + s_1) + (t_2 + s_2)k_o + t_3 k_o k_i$$

The large outer loop term gives rise to highly asymmetric resource contours, as shown in

Figure 6.2. In contrast to the previous chapter, here the linear term is very significant and can dominate. Thus, resource concerns lend a strong bias toward inner loop unfolding. Both parallelism and resource concerns suggest that outer loop unfolding should be minimized in the presence of outer loop structure recurrences.

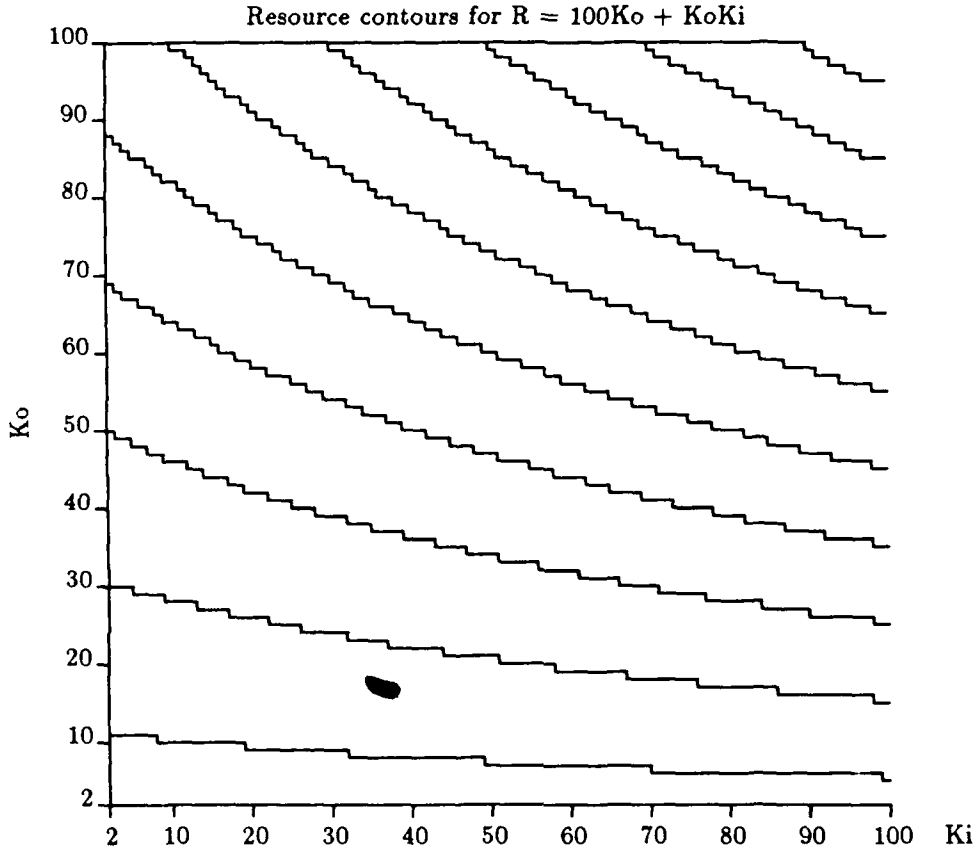


Figure 6.2: Typical Combined Token and Structure Storage Contours

6.2 An Example: Successive Relaxation

To study the behavior of nested loops with structure recurrences, we examine a program which performs successive relaxations of a mesh using the **relax_step'** function discussed in the previous chapter. (Structures can be recycled in the outer loop, as discussed in Chapter 3, however, we have not done so, in order to control the experiment. Recycling structures introduces an additional loop to clear the structure before it is reused, causing the topology to fall outside the

k_o	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	80	112	147	165	188	214	251	253	258	302	305	308	311	317	457
15	80	112	147	165	188	214	251	253	258	302	305	308	311	317	457
14	80	112	147	165	188	214	251	253	258	302	304	307	310	314	450
13	76	112	133	163	188	211	215	253	258	294	297	299	302	305	433
12	75	103	132	161	183	207	211	245	258	287	289	292	295	298	422
11	71	102	130	159	180	204	207	239	258	280	283	285	287	290	407
10	71	101	129	143	177	200	203	234	239	275	278	280	282	285	397
9	67	94	118	142	158	178	199	205	234	269	271	274	276	278	385
8	67	93	117	140	156	175	197	201	229	233	267	269	271	274	376
7	55	75	102	111	143	145	177	197	202	204	205	207	209	212	305
6	51	71	89	109	129	141	144	175	195	197	198	200	202	204	290
5	43	57	83	84	109	110	140	153	154	156	157	159	160	163	236
4	39	54	70	82	99	107	116	136	148	150	151	152	154	156	222
3	29	38	55	56	72	73	97	102	103	104	105	106	107	109	158
2	21	28	40	42	54	55	72	76	77	78	78	79	80	81	117

k_m

Figure 6.3: P_{ave} Throughout the k-Space for Relax 16 Steps on 16 × 16 Mesh

perfectly nested loop formulations we are considering so far.) The calling tree for this example is shown below. Note, that this version of the relaxation program uses conditionals, rather than separate loops, to handle boundary conditions, so it has a strict nested loop structure.

Although the natural unfolding for the innermost loop is 5, in the following we have chosen to set it at 2 and explore only the effects of the outer and middle loop bounds. (The additional parallelism available by allowing the innermost loop to unfold further is small, as indicated by Figure 5.19, above.) In this example, the structure storage coefficient, s_2, is n^2, while t_3 is a small constant. Thus, even a small amount of outer-loop unfolding causes a large increase in the overall resource requirement. The average parallelism over the k-space for this program is shown in Figure 6.3. Observe that for large values of k_o there is almost no improvement in P_{ave}. It appears here that μ_o is less than 16, due to the structure recurrence. There is no change in the top two rows. Generally, more parallelism is exposed by allowing the middle loop to unfold, rather than the outer loop. This is in direct opposition with the behavior observed for the simple loops studied in the previous chapter.

k_o	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	80	112	147	165	188	214	251	253	258	302	305	308	311	317	457
15	80	112	147	165	188	214	251	253	258	302	305	308	311	317	457
14	80	112	147	165	188	214	251	253	258	302	304	307	310	314	450
13	76	112	133	163	188	211	215	253	258	294	297	299	302	305	433
12	75	103	132	161	183	207	211	245	258	287	289	292	295	298	422
11	71	102	130	159	180	204	207	239	258	280	283	285	287	290	407
10	71	101	129	143	177	200	203	234	239	275	278	280	282	285	397
9	67	94	118	142	158	178	199	205	234	269	271	274	276	278	385
8	67	93	117	140	156	175	197	201	229	233	267	269	271	274	376
7	55	75	102	111	143	145	177	197	202	204	205	207	209	212	305
6	51	71	89	109	129	141	144	175	195	197	198	200	202	204	290
5	43	57	83	84	109	110	140	153	154	156	157	159	160	163	236
4	39	54	70	82	99	107	116	136	148	150	151	152	154	156	222
3	29	38	55	56	72	73	97	102	103	104	105	106	107	109	158
2	21	28	40	42	54	55	72	76	77	78	78	79	80	81	117

k_m

Figure 6.4: W_{max} Optimal Points over the k-Space for Relax, Shown Overlaying P_{ave} Values

6.3 Local-Storage Limited Policies

To contrast the behavior of various policies in the context of structure recurrences with that observed in the previous chapter, we first consider only local storage, ignoring structure storage, as we did there. Later, we consider the additional impact of the structure storage constraint.

The W_{max} optimal k-configurations for this program are shown as boxed entries in Figure 6.4. Observe, that all of these fall into the region $k_o < k_m$. Thus, we can expect the Outer-In policies to perform poorly. Fitting the measured token storage in Figure 6.5 to our general resource expression we get

$$T(k_o, k_m) = 20 + 7k_o + 16.5k_o k_m.$$

This fits well for small bounds, but over estimates the token storage by 50% at the point of maximum unfolding. We will use this resource expression in implementing our resource-limited policies.

Figure 6.6 shows a summary of the performance of our five policies under various waiting token limits on Relax (16×16 for 16 steps). The three outer loop oriented policies (indicated by $\bigcirc$, $+$, and $\times$) perform poorly, even ignoring structure storage usage, because of lack of useful unfolding in the outer loop. The Diagonal (squares) and Inner-Out (triangles) policies,

k_o	k_m = 2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0	2.1	2.1	2.2	2.3	2.4	2.6	2.8
15	0.5	0.7	0.9	1.1	1.3	1.5	1.7	1.9	1.9	2.0	2.1	2.3	2.4	2.5	2.7
14	0.5	0.7	0.9	1.0	1.2	1.4	1.6	1.7	1.8	1.9	2.0	2.2	2.3	2.5	2.7
13	0.5	0.7	0.8	1.0	1.2	1.3	1.5	1.6	1.7	1.8	2.0	2.1	2.3	2.4	2.6
12	0.5	0.6	0.8	0.9	1.1	1.2	1.4	1.5	1.6	1.7	1.9	2.0	2.1	2.3	2.5
11	0.4	0.6	0.7	0.9	1.0	1.2	1.3	1.4	1.5	1.6	1.8	1.9	2.0	2.2	2.4
10	0.4	0.5	0.7	0.8	0.9	1.1	1.2	1.3	1.4	1.5	1.7	1.8	1.9	2.0	2.2
9	0.4	0.5	0.6	0.7	0.8	1.0	1.1	1.2	1.3	1.4	1.5	1.7	1.8	1.9	2.1
8	0.3	0.4	0.5	0.7	0.8	0.9	1.0	1.1	1.1	1.3	1.4	1.5	1.6	1.7	1.9
7	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.9	1.0	1.1	1.2	1.3	1.4	1.5	1.6
6	0.3	0.4	0.4	0.5	0.6	0.7	0.8	0.8	0.9	1.0	1.1	1.1	1.2	1.3	1.4
5	0.2	0.3	0.4	0.4	0.5	0.6	0.6	0.7	0.8	0.8	0.9	1.0	1.0	1.1	1.2
4	0.2	0.2	0.3	0.4	0.4	0.4	0.5	0.6	0.6	0.7	0.7	0.8	0.8	0.9	1.0
3	0.1	0.2	0.2	0.3	0.3	0.4	0.4	0.4	0.5	0.5	0.6	0.6	0.6	0.7	0.7
2	0.1	0.1	0.2	0.2	0.2	0.2	0.3	0.3	0.3	0.3	0.4	0.4	0.4	0.5	0.5

Figure 6.5: $W_{max}/1000$ for Relax 16 steps on 16×16 Mesh

perform substantially better than the Outer-In family, and the shortfall at larger resource levels can be attributed to overestimating the token requirement. Referring to Figure 6.4, one can expect that a Jog&Jump-Inner-Out policy, *i.e.*, a slight bias toward outer loop unfolding under moderate resource levels, should perform quite well.

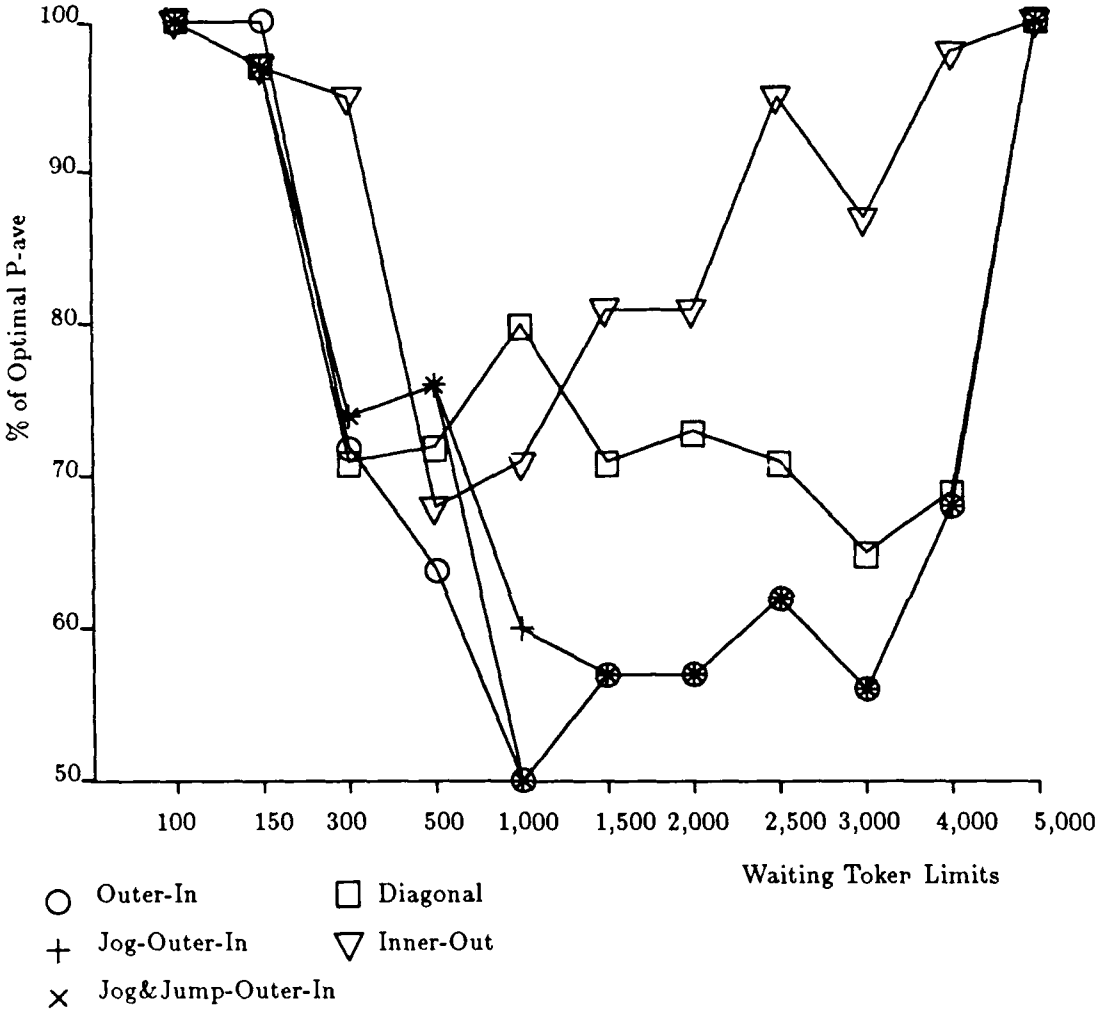


Figure 6.6: % Optimal P_{ave} for Five Waiting Token Limited Policies on Relax 16×16 for 16 steps

W	k_o	k_m	W_{max}	W_{max}/W	P_{ave}	Opt	P_{ave}/Opt
100	2	2	100	1.00	21	21	1.00
150	3	2	142	0.95	29	29	1.00
300	4	3	243	0.81	54	76	0.71
500	4	6	395	0.79	99	117	0.85
1,000	4	14	848	0.85	154	222	0.69
1,500	4	16	956	0.64	222	290	0.77

Figure 6.7: Outer-In Token-Limited Policy with 4 Structures

W	k_o	k_m	W_{max}	W_{max}/W	P_{ave}	Opt	P_{ave}/Opt
100	2	2	100	1.00	21	21	1.00
150	2	3	142	0.95	28	29	0.97
300	3	5	281	0.94	56	76	0.74
500	4	6	395	0.79	99	117	0.85
1,000	4	14	848	0.85	154	222	0.69
1,500	4	16	956	0.64	222	290	0.77

Figure 6.8: Jog-Outer-In Token-Limited Policy with 4 Structures

W	k_o	k_m	W_{max}	W_{max}/W	P_{ave}	Opt	P_{ave}/Opt
100	2	2	100	1.00	21	21	1.00
150	2	3	142	0.95	28	29	0.97
300	2	8	278	0.93	72	76	0.95
500	2	14	441	0.88	80	117	0.68
1,000	3	16	720	0.72	158	222	0.71
1,500	4	16	956	0.64	222	290	0.77

Figure 6.9: Inner-Out Token-Limited Policy with 4 Structures

6.4 Separate Structure and Token Storage Constraints

We have considered the effects of the structure recurrence only in so far as it changes the parallelism contours, ignoring structure storage resources. A structure storage constraint, separate from the constraint on token storage, places a hard limit on k_o. Thus, the effect of such an approach can be seen by simply limiting the maximum unfolding of the outer loop. This has the collateral effect of pruning the region of the k-space in a way that avoids the excessive unfolding of the outer loop. As a result, all of the policies perform better relative to the optimal k-configuration, as can seen in Figures 6.7, 6.8, and 6.9.

k_o	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	80	112	147	165	188	214	251	253	258	302	305	308	311	317	457
15	80	112	147	165	188	214	251	253	258	302	305	308	311	317	457
14	80	112	147	165	188	214	251	253	258	302	304	307	310	314	450
13	76	112	133	163	188	211	215	253	258	294	297	299	302	305	433
12	75	103	132	161	183	207	211	245	258	287	289	292	295	298	422
11	71	102	130	159	180	204	207	239	258	280	283	285	287	290	407
10	71	101	129	143	177	200	203	234	239	275	278	280	282	285	397
9	67	94	118	142	158	178	199	205	234	269	271	274	276	278	385
8	67	93	117	140	156	175	197	201	229	233	267	269	271	274	376
7	55	75	102	111	143	145	177	197	202	204	205	207	209	212	305
6	51	71	89	109	129	141	144	175	195	197	198	200	202	204	290
5	43	57	83	84	109	110	140	153	154	156	157	159	160	163	236
4	39	54	70	82	99	107	116	136	148	150	151	152	154	156	222
3	29	38	55	56	72	73	97	102	103	104	105	106	107	109	158
2	**21**	**28**	**40**	**42**	**54**	**55**	**72**	**76**	**77**	**78**	78	**79**	**80**	**81**	117

k_m

Figure 6.10: Optimal Points in k-Space Under Combined Storage Metric

6.5 Combined-Storage Limited Policies

If structure storage and local storage are drawn from the same pool, it makes sense to consider the combined storage usage. This raises the interesting possibility that parallelism may be improved by trading a version of the data structure for more local token storage, or vice-versa. The combined resource usage for a k_o, k_m configuration of an $n \times n$ Relax can be approximated by

$$R(k_o, k_m) = 20 + (7 + n^2)k_o + 16.5k_o k_m.$$

The optimal k-configurations under this resource expression are shown by boxed entries in Figure 6.10. The best policy in this case clearly is Inner-Out. The behavior of four policies is shown in Figure 6.11. Here the family of Outer-In policies performs very poorly, and Inner-Out is perfect! Reducing the amount of local storage to allow for more versions of the structure is seldom advantageous. However, a non-trivial amount of parallelism can be obtained from outer loop unfolding, if there is sufficient resource availability.

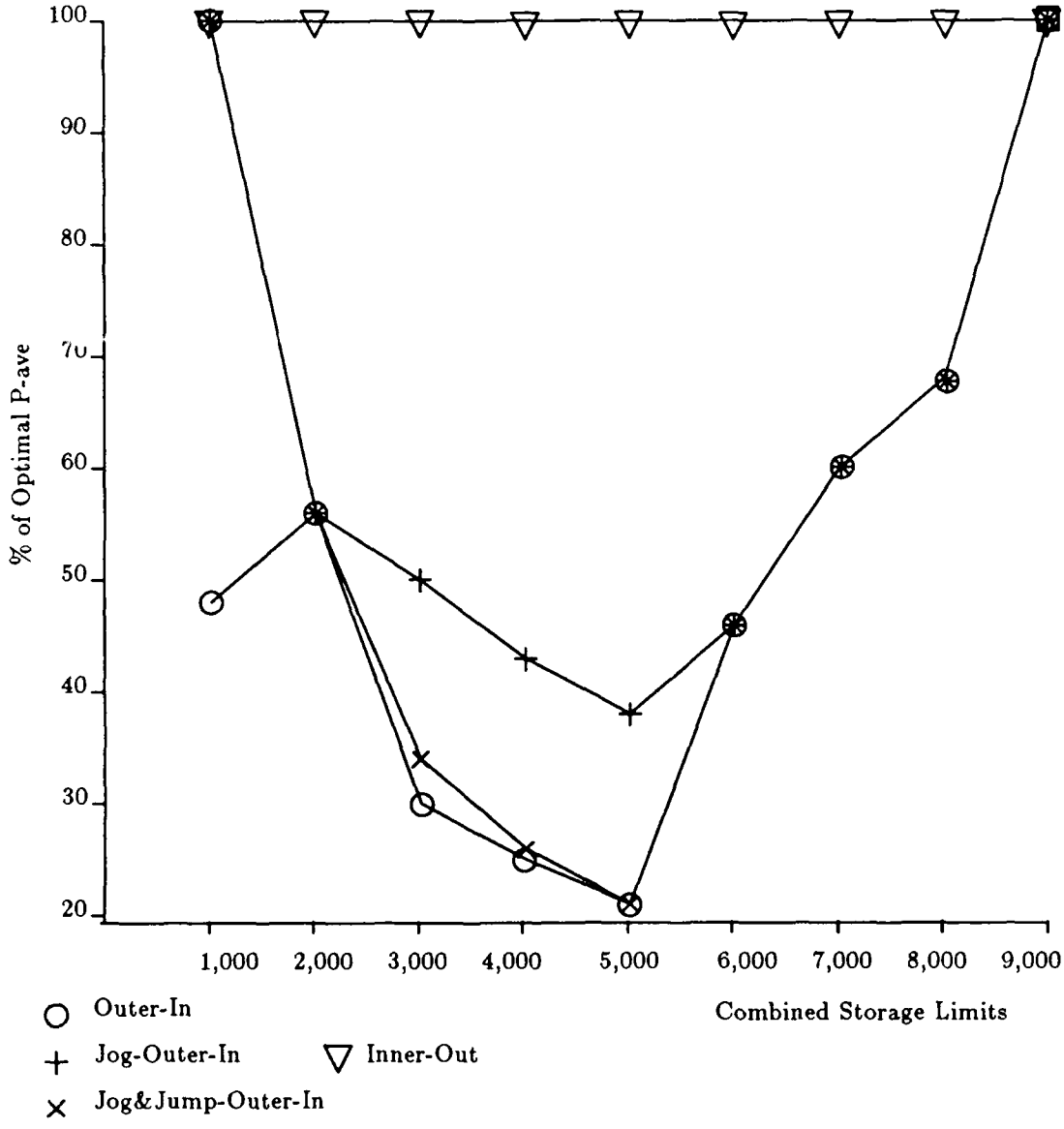


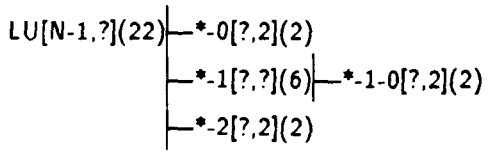
Figure 6.11: % Optimal P_{ave} for Five Combined Storage Limited Policies on Relax 16 × 16 for 16 steps

6.6 Variations

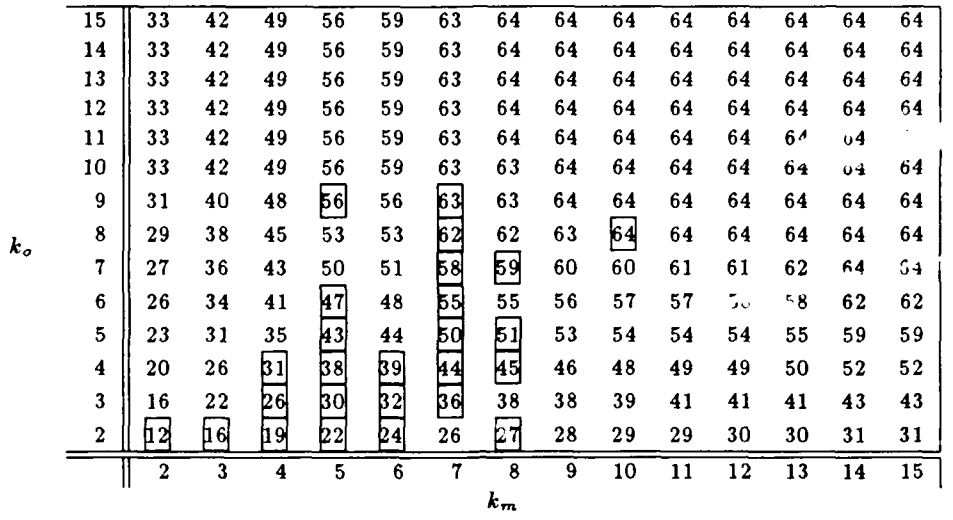
To test further the guidelines developed above, we consider a variation involving dynamic loop bounds. The example we consider is LU decomposition, without pivoting. The *Id* code is shown below. Like the previous example, there is a structure recurrence in the outer loop, but in addition the iteration counts of the other loops decrease with the outer loop index.

```
def LU A n =
  {D = i_matrix ((1,n),(1,n));
   {for k <- 1 to n-1 do
      {for j <- k to n do                    % copy pivot row (U)
         D[k,j] = A[k,j]};
      {for i <- k+1 to n do                  % multipliers (L)
         D[i,k] = a[i,k]/a[k,k]};
      next A = I_matrix ((k+1,n),(k+1,n));  % eliminate
      {for i <- k+1 to n do
         {for j <- k+1 to n do
            next A[i,j] = A[i,j] - D[k,j]*D[i,k]}}
    finally {d[n,n]=a[n,n]}};
    in d};
```

The calling tree for this program is shown below. It shows the branching within the outer loop apparent from the *Id* code. Technically, this falls outside the strict nested loop formulation, but the three innermost loops have bounded unfolding, as determined by the path-length analysis outlined in Chapter 4. Ignoring these, we are left with a strict loop nest.



The experiment we conduct is to bound the three innermost loops to 2 and study the effects of the two remaining k-bound parameters — the outermost loop and the middle loop. The results of this study are presented in Figure 6.12, which shows P_{ave} over the remaining k-space for a 15×15 matrix. To draw a comparison with the results in the previous chapter, we consider only local storage, in this case activation frames. A_{max} optimal points are indicated by boxes. Here the Inner-Out policy is difficult to implement, since the iteration count of the middle loop depends upon the outer loop index. If the average iteration count were treated as the maximum unfolding, *i.e.*, $k_m < 8$, such a policy would do well, but so would a diagonal policy.



k_o \ k_m	2	3	4	5	6	7	8	9	10	11	12	13	14	15
15	33	42	49	56	59	63	64	64	64	64	64	64	64	64
14	33	42	49	56	59	63	64	64	64	64	64	64	64	64
13	33	42	49	56	59	63	64	64	64	64	64	64	64	64
12	33	42	49	56	59	63	64	64	64	64	64	64	64	64
11	33	42	49	56	59	63	64	64	64	64	64	64	64	
10	33	42	49	56	59	63	63	64	64	64	64	64	64	64
9	31	40	48	56	56	63	63	64	64	64	64	64	64	64
8	29	38	45	53	53	62	62	63	64	64	64	64	64	64
7	27	36	43	50	51	58	59	60	60	61	61	62	64	64
6	26	34	41	47	48	55	55	56	57	57	58	62	62	
5	23	31	35	43	44	50	51	53	54	54	54	55	59	59
4	20	26	31	38	39	44	45	46	48	49	49	50	52	52
3	16	22	26	30	32	36	38	38	39	41	41	41	43	43
2	12	16	19	22	24	26	27	28	29	29	30	30	31	31

Figure 6.12: P_{ave} for LU decomposition on 15×15 Matrix with A_{max} Optimal Points Marked

6.7 Conclusions

In the presence of an outer loop recurrence on a structure, Outer-In policies generally perform worse than Inner-Out policies, although independent I-structure constraints may keep outer loop unfolding small enough that Outer-In policies still perform reasonably well. This stands in contrast to the kind of nest examined in the previous chapter, where the nest possessed only very weak recurrences. These results suggest a simple loop classification scheme:

1. **Limited Useful Unfolding (LUU):** Small innermost loops, loops with very strong recurrences, and loops with small constant iteration counts will have limited useful unfolding, regardless of problem size or resource availability. They can largely be ignored from a policy viewpoint. Path-length analysis on the loop graph will detect the first group, *i.e.*, c is small. Identifying loops of the second group requires more substantial dependence analysis, possibly including subscript analysis and inter-procedural strictness analysis. The goal is basically to show that $i \approx c$. The last group often can be detected through symbolic analysis of loop initial, step, and final expressions.

2. **Unfold if Necessary (UNIN):** Loops with large resource coefficients, such as the outer loop of Relax, may offer parallelism when allowed to unfold, but at a large resource investment. If parallelism can be exposed elsewhere, it should be. To place a loop into

this category involves determining where resources are allocated and is strongly related to determining the lifetime of data structures. In many cases a loop that properly belongs in the first category, but is not identified as such because the dependencies are hard to trace, will often be placed in this category. However, the weaker categorization is usually sufficient to arrive at a reasonable policy.

3. **Unfold if Possible (UIP):** Loops like the outer loop in Matrix Multiply that offer ample unfolding with relatively modest resource investment. In the absence of further information, we assume a loop is of this form.

Our Relax example offers an interesting illustration of these categories. The innermost loop falls in the first category, with maximum unfolding (in the absence of latency) of 5. Resource analysis places the outermost loop in the second category; it is possible that it belongs in the first, but, fortunately, it is enough to mark it as unattractive. The middle loop falls in the last category.

One can construct loops that have a fairly strong recurrence from one iteration to the next (large i) that does not involve the bulk of the computation in the body, so that i is independent of c. This would not fall well into any of our categories, and detailed analysis is required to determine which loop actually admits more parallelism. However, this sort of situation is fairly rare in practice, and the net difference between the policy alternatives is small. It appears that the bulk of loops encountered in practice fall well into one category or another, the caveat being the deep dependencies mentioned above.

Chapter 7

Controlling Adjacent Loops

The nested loops studied in the previous chapters represent a special kind of branching in the execution tree, where the branch factor is dynamically determined — the k-bound. The policy for choosing k-bounds dictates the shape of the subtree rooted at the outer loop, such that the total resource requirement of the nest does not exceed some value. The other kind of execution tree branching arises from multiple invocations within a code-block or iteration. In general, we cannot order such invocations, but must let them proceed in parallel. In this case, the branching factor is static, and the k-bound assignment for loops in the two branches dictate how resources are partitioned at the branch. Thus, the primary issue is to decide how a given amount of resources are to be divided between the branches; once the division is determined, we can apply the policy guidelines recursively in each branch. What makes this decision difficult is the potential variation in the relationships of *adjacent* loops, *i.e.*, loops appearing on different branches of the calling tree.

In this chapter, we explore these various relationships in some detail and argue for a general approach of proportional allocation. Based on this idea and the theory developed in Chapter 4, a policy of *threshold splitting* is formulated. Empirical studies confirm that the optimal resource division changes as the amount of available resources increases, and that threshold splitting handles this nicely. This policy is then carried out on several examples.

7.1 Loop Relationships

We begin the study of adjacent loops by examining various potential relationships between adjacent loop nests. Generally, it is easier to determine the relative sizes of the loop nests than

their actual interactions. We arrive at a basic notion of proportional allocation: give more to the larger computation.

7.1.1 Serial loops

The simplest relationship from a policy viewpoint is one in which the branches are completely serialized, *i.e.*, the computation in one branch must complete before that in the other begins. All available resources are simply provided to each branch in turn. In general, however, it is not possible to serialize a pair of loops[86] and, when it is possible, assessing the cost/performance trade-offs is extremely difficult. Costs include the overhead due to the barrier and the loss of potential parallelism. What makes this so difficult to assess is that it may depend on the actual arrival patterns of data, which is a global phenomenon. On the other hand, under resource-limited execution the performance improvement derived from giving full resources to each phase, rather than partitioning resources between them, may be significant. Although we recognize serialization as a potentially interesting line of study, we do not pursue it here. The `SEQ` construct in *Id* provides a means of introducing serialization, and one may assume that a separate program transformation might introduce `SEQ` where it is safe and beneficial. We will take advantage of such serialization where it arises, but not introduce it ourselves.

7.1.2 Independent loops

If two adjacent loop nests are completely independent, the goal of resource splitting is to end up with essentially equal critical path lengths. Suppose, on the contrary, that we have two adjacent loop nests with differing τ_∞. Since the critical path of a loop nest is monotonically non-increasing with increasing resources, allocating more resources to the one with the larger τ_∞ reduces its critical path at the expense of increasing the critical path of the other. A minimum is achieved where they are equal. If the loop nests were identical, this would mean giving them equal resources. Otherwise, we must give more resources to the larger computation, so that it will complete as fast as the smaller.[1]

[1] We note that the other possible minimum occurs where one loop is allowed to finish before the other begins, so each have the full resource capability to work with, as discussed above.

7.1.3 Producer-consumer loops

Optimal partitioning of resources among producer-consumer relationships is difficult to characterize, because these relationships can be so complex. If we imagine a wavefront of values flowing from the producer to the consumer, possibly via I-structures, then the goal is essentially to match the rates of production and consumption. If too little of the resources are given to the producer, then those allotted to the consumer may be squandered for lack of data to operate on. If too much is allocated to the producer, the consumer may be unnecessarily constrained and fail to keep pace. This again suggests proportional allocation.

However, this wavefront model may be inappropriate in many circumstances. For example, the consumer may require a substantial portion of the intermediate structure before it can make significant progress, in the worst case requiring just the last few values produced. Alternatively, suppose the complexity of the producer is much smaller than that of the consumer, as might occur when the producer initializes and the consumer transforms a data structure. Then, for a short period, the resource load is the sum of the two loop nests, but once the producer completes, the portion of the resources allocated to it go unused. They cannot be given back to the consumer within our current scheme.

7.1.4 Cooperative loops

Finally, we must consider cases where pairs of loop nests are mutually recursive — neither can proceed without the other. In this scenario, constraining either loop will slow the progress of the other, so resources should be partitioned so that progress is even. There is no guarantee that this is possible, as the production of shared data may be non-uniform, but one can expect that the larger computation will require more of the resources.

7.2 Proportional Allocation

The common thread in all these adjacent loop scenarios is the notion of proportional allocation — we should give more resources to the more complex computation. The question is, What should be the proportion, and how does this change with the availability of resources, size of problem, or structure of the computation?

Suppose we have a program with two adjacent doubly nested loops, bounded by k_{11}, k_{12}

and k_{21}, k_{22}, respectively. As argued in Chapter 4, the worst case resource requirements are of the form

$$R(k_{11}, k_{12}, k_{21}, k_{22}) = k_{11}(r_{11} + k_{12}r_{12}) + k_{21}(r_{21} + k_{22}r_{22}).$$

Thus, a given assignment of the k-bounds determines the fraction of the total resources used by each nest. For example, if the k-bounds for the two nests are similar, the nest with the larger resource coefficients will use more of the resources. Whereas for nested loops the policy question is *which loop level to favor*, for adjacent loops the question is which branch to favor and how much so. We may expect that the policy will have to be sensitive to the level of resource availability, as well as program structure. As for nested loops, a given policy must be translated into a simple rule for computing the k-bounds.

As a first step in formulating a policy for dealing with adjacent loops, we examine the situations in which the choice of k-bounds is clear, and then we consider ways of handling the intermediate, less obvious, situations. Suppose resources are sufficiently scarce that all k-bounds must be set to the minimum value. Then, the resource division at a given branch point is completely determined by resource requirements of the various subtrees. We estimate these requirements as follows. Having formed the static invocation tree, work up from the leaves computing, for each node v, the resource requirement of the subtree rooted at v with all loops tightly bound, $R_{min}(v)$. As discussed in Chapter 4, the requirement for a loop node is essentially k times the requirement per iteration and for a branch node it is the requirement of the node plus the sum of the branches. The minimum activation frame requirement of each subtree is shown in parentheses in the call trees above. Now, consider a branch node v with children $v_1, \ldots, v_n$. The fraction of the resources available at v that are assigned to branch i is

$$f(i) = \frac{R_{min}(v_i)}{\sum_{j=1}^{n} R_{min}(v_j)}.$$

So, whatever policy we implement for partitioning resources at a branch, it should degrade to this division when resources are very scarce.

At the other extreme, we may ask what is the optimal fraction when resources are plentiful. If all the loops are allowed to unfold completely, the resource partition at a given branch point is determined by the requirements of the subtrees rooted there. Suppose we compute, for each node v, $R_{max}(v)$, the resource requirement for the subtree rooted at v with all k-bounds set to their maximum value, *i.e.*, the iteration count for the loop. Of course, for most loops this can

only be determined symbolically and for some loops it cannot be determined at all, but let us ignore these problems for the moment. For branch node v with children $v_1, \ldots, v_n$, the fraction of the available resources at v that are assigned to branch i is

$$F(i) = \frac{R_{max}(v_i)}{\sum_{j=1}^{n} R_{max}(v_j)}.$$

Thus, the fraction of the resources that should be allotted to each branch is clear at some very tight resource level, $R_{min} = \sum R_{min}(v_j)$, and at some very plentiful level, $R_{max} = \sum R_{max}(v_j)$. The heart of the policy question is what to do between these levels.

Figure 7.1 shows three possible splitting policies, assuming that there are two branches with significantly different complexity. The horizontal axis is the total resource constraint, r, running from R_{min} to R_{max}. The vertical axis is the amount of resources allocated to a branch. At $r = R_{min}$, branch i must receive $R_{min}(i)$, so the fraction of resources assigned to branch 1 is $f(1)$. Ideally, at $r = R_{max}$ branch i should receive $R_{max}(i)$. A policy will dictate $R_r(1)$ and $R_r(2)$ for intermediate resource levels r, under the condition $R_r(1) + R_r(2) = r$.

The thin lines show the effect of *extrapolating* the ratio $f(i) = \frac{R_{min}(i)}{R_{min}}$ throughout the resource spectrum. This policy is easy to implement, because we can always compute R_{min} ratios for local storage. However, observe that after a certain point, the smaller branch has completely unfolded and the additional resources allocated to it go to waste, while the larger branch is constrained throughout.

A second choice, shown in thick lines, is to *interpolate* between the values at the resource extremes. This avoids squandering resources in the manner of the extrapolation rule, but presents two other difficulties. First and foremost, we may not be able to determine $R_{max}(i)$ for some branches. Second, the smaller loop is likely to be in the partial-wave plateau region throughout most of the spectrum. A small amount of additional resources would allow it to unfold completely.

The third possibility, shown in dashed lines, is a combination of the other two. The idea is that the branches representing the smaller computations are likely to be the ones where R_{max} can be determined, at least symbolically. The R_{min} ratio is employed until $R_r(1) = R_{max}(1)$ is reached for branch 1, the smaller of the branches. From that point on, the resource allocation for the saturated loop nest is held steady and all remaining resources are assigned to the other branch. More generally, if we have a number of branches, where R_{max} is known for some, but not

all, the *threshold splitting policy* first allocates resources to the ones for which R_{max} is known, based on R_{min} ratios and by increasing R_{max}. The remaining resources are divided between the remaining branches according to their R_{min} ratios. Often, many of the uninteresting branches can be dismissed by this simple strategy.

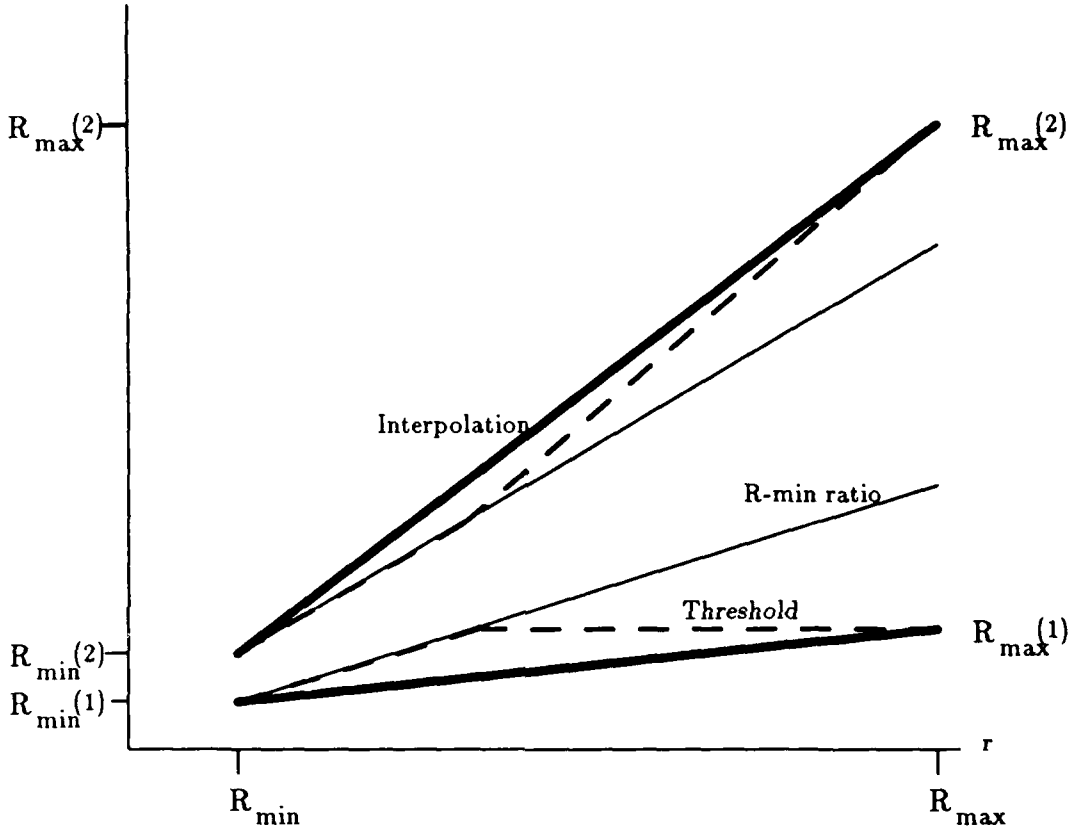


Figure 7.1: Three Possible Splitting Policies

7.3 Experiments with Resource Partitioning

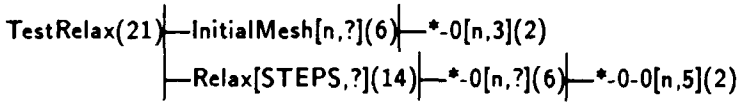
To gain a deeper understanding of the behavior of adjacent loops, we examine several examples. This study requires a somewhat different empirical method than was employed for studying nested loops. Here the space of possibilities is large, as it is basically the cross product of the k-bounds for each of the loops on each branch. The key to focusing our attention on the interesting part of this space is first to fix the loop-bounding policy for each branch and then study the behavior of the whole as a function of the resource level, r, and the resource partition,

f. The assumption here is that the best policy for a loop nest is not strongly affected by the context in which it appears.

Below, we follow this methodology for several kinds of adjacent loops. We look at two producer-consumer examples with differing characteristics. The first is based on the **Relax** program studied previously; it initializes a matrix as one phase and performs successive relaxations as the other. In this example, there is a natural overlap between the two phases, as the wavefront produced by one becomes the first wavefront of the other. The second example initializes a matrix and squares it. Here the overlap between the two phases is harder to predict, as the consumer accesses the intermediate matrix in two different ways and much of it has to be defined before the consumer can produce any results. We then examine branching due to special boundary conditions, and, finally, branching due to heterogeneous computational phases. In each case, we consider activation frame usage and structure usage separately.

7.3.1 Producer/consumer: initialize and relax

Here we study a producer/consumer pair of loop nests, which initialize and relax a mesh, respectively. The calling tree is:



The initialization portion is a doubly nested loop, where the inner loop has bounded useful unfolding. The Relax loop is the problem we studied in Chapter 6, and accordingly we adopt the Inner-Out policy for the middle and outer loops, with the innermost loop bounded to 2. The augmented program, with code to determine loop bounds from activation frame limits, is shown below. The augmented form of **relax_step** simply has two new arguments, the k-bounds for the two loop levels. At the top-level where code to determine the split should be, we have introduced an extra parameter, **f**, so that we can study this policy aspect.

```
def TestRelax n steps R f =
  {frames = R - 1;
   init_frames = floor (frames * f);
   relax_frames = frames - init_frames;
   A = InitialMesh n init_frames;
   in Relax A steps relax_frames};

% Frame limited InitialMesh (Outer Unfolds)
def InitialMesh n frames =
  {A = i_matrix ((1,n),(1,n))
    {FOR i <- 1 to n bound (max 2 (min n (floor (frames / 3)))) DO
       {FOR j <- 1 to n bound 2 DO
          A[i,j] = iv i j n}}
    in A};

% Frame-limited Relax (Inner-out policy)
def Relax A steps frames =
  {((l1,u1),(l2,u2)) = (2D_bounds A);
   ki = 2;
   km = (max 2 (min (u2 - l2 + 1) (floor ((frames - 2) / 6))));
   ko = (max 2 (min steps (floor (frames / (2 * km)))));
   in {for i <- 1 to steps bound ko do
         next A = call relax_step' A  km ki;
       finally A}};
```

Here the producer (**InitialMesh**) has the same complexity as a single iteration of the outer loop of the consumer (**Relax**). This raises concern over possibly squandering resources once the producer terminates. However, the overlap between the two loop nests is more complete than one might expect. Figure 7.2 shows parallelism profiles for the two phases and their composition under complete unfolding. First, we see a bell-shaped curve for **initial_mesh**, with $n = 16$. This has $\tau_\infty = 206$. Above this is the curve for the **relax**, with *steps* $= n$. It has a $\tau_\infty = 615$. The uppermost curve shows the profile for the composition of the two, **testrelax**. τ_∞ is only 637.

Figure 7.3 shows the behavior of **testrelax** for 16 steps on a 16×16 matrix at several resources partitions, f (the fraction of resources assigned to **initial_mesh**), over a range of activation frame resource levels. Each entry shows P_{ave} and below it, in parentheses, the maximum number of activation frames in use at any point in the ideal execution. The fraction yielding maximal P_{ave}, indicated by entries in bold, is near 0.2 at 200 activation frames and below, but starts falling somewhere beyond this point.

The R_{min} rule would put $f = \frac{6}{6+14} = 0.3$, slightly higher than the observed optimum under tight resource constraints and considerably higher than that observed under looser ones. This deserves some explanation. The r_{min} ratio is required at $A = 21$, because this reflects the

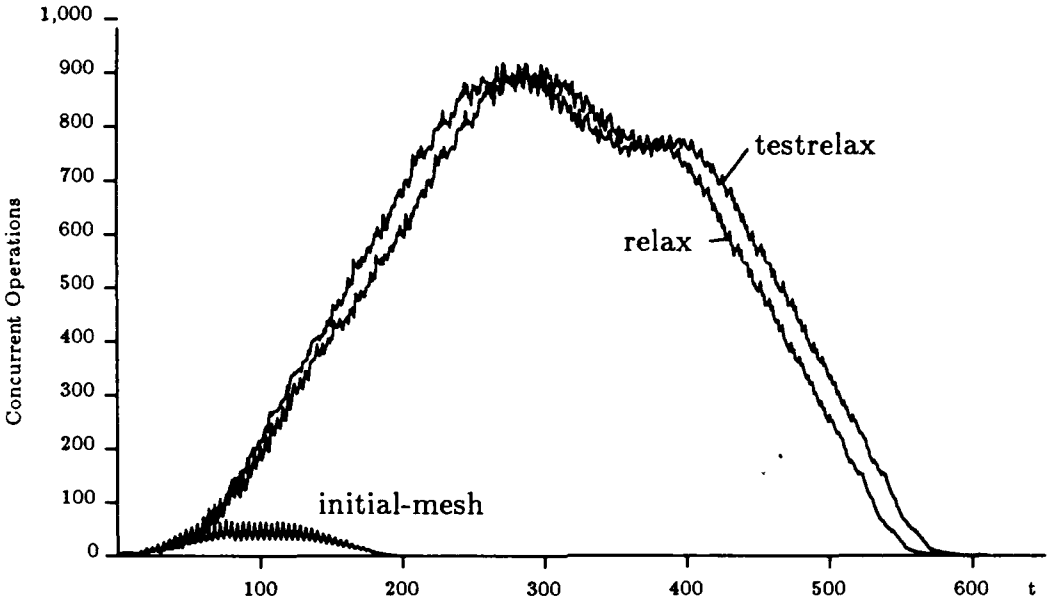


Figure 7.2: Parallelism Profiles for Two Phases of TestRelax and Their Composition

$f =$	0.05	0.1	0.2	0.3	0.4	0.5
$A = 50$	51	51	**53**	43	41	29
	(50)	(50)	(47)	(44)	(44)	(44)
$A = 100$	69	73	**79**	78	77	73
	(98)	(95)	(98)	(95)	(95)	(98)
$A = 200$	122	145	**155**	118	118	118
	(156)	(165)	(177)	(140)	(140)	(140)
$A = 400$	257	**290**	285	234	220	220
	(361)	(352)	(295)	(257)	(219)	(219)
$A = 800$	421	**435**	419	396	374	366
	(635)	(611)	(585)	(518)	(438)	(392)
$A = 1200$	**442**	**442**	**442**	**442**	435	409
	(653)	(653)	(653)	(653)	(611)	(553)

Figure 7.3: $P_{ave}(A_{max})$ Versus Fraction Allocated to Smaller Loop Nest for Several Resource Limits in Relax.

	f	0.05	0.1	0.2	0.3	0.4	0.5
A	200	122	145	155	**158**	**158**	**158**
		(156)	(165)	(177)	(181)	(181)	(181)
A	400	257	290	**300**	**300**	**300**	**300**
		(361)	(352)	(343)	(343)	(343)	(343)
A	800	421	**442**	**442**	**442**	**442**	**442**
		(635)	(634)	(634)	(634)	(634)	(634)
A	1200	**442**	**442**	**442**	**442**	**442**	**442**
		(653)	(653)	(653)	(653)	(653)	(653)

Figure 7.4: $P_{ave}(A_{max})$ Versus *Limited* Fraction Allocated To Smaller Loop Nest for Several Resource Limits in Relax.

number of activation frames required by each branch with all the k-bounds set to the minimum value, 2. Above this resource level, the R_{min} ratio is not necessarily optimal, as it does not reflect the true size of the two computations. At $R = 50$ the linear interpolation rule would give $f \approx 0.14$; f drops rapidly from there.

At small values of r and f there is a second factor influencing the results. The minimum number of activation frames required by the initialization branch is 6. Thus, at $R = 50$ and $f = 0.05$ we effectively overcommit the resources, since the initialization portion is assigned 2 but uses 6 and the relax branch is assigned 48. The observed maximum activation frame usage is 50 in this case, so under the ideal model we just got by. This phenomenon represents a defect in the experimental method and influences only a few entries in the upper-left corner of the figure. Under tight resource constraints it causes small values of f to appear artificially good.

The maximum usable resource threshold for the initialization portion is 48 frames, with $n = 16$. Thus, we can expect that for $Af > 48$ resources are wasted on this branch. In our experiment, this plays a significant role for $A \geq 200$. To bring the effect of this threshold into the picture, we change our policy so that no more than $3n$ activation frames are assigned to the initialization branch and the remainder is assigned to the other. This has no effect for the first two rows, but for the other resource levels we get the behavior shown in Figure 7.4. The R_{min} proportion would assign 30% of the resources to the producer. We see that with the threshold in effect this is optimal nearly throughout.

7.3.2 Producer/consumer: initialize and square

In this section, we study another example with two branches representing distinct computational phases, a doubly-nested producer and a triply-nested consumer, but the relationship between the two phases is quite different here. Instead of the initial matrix serving as the first wave of a sequence of transformations, the consumer accesses the initial matrix in complex ways. Surprisingly, the overlap between the two phases is quite significant, as shown in Figure 7.5. Here τ_∞ for the producer, consumer, and composition are 211, 298, and 310, respectively.

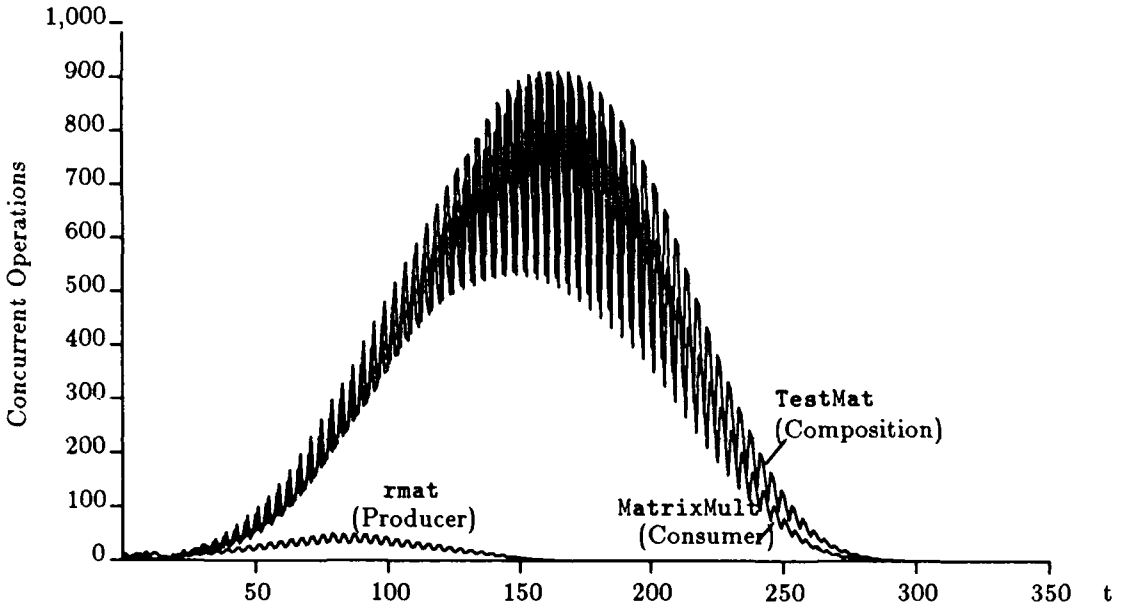
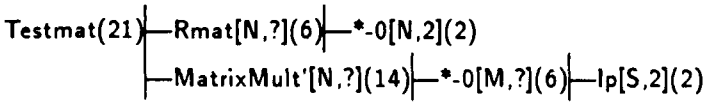


Figure 7.5: Parallelism Profiles for Two Phases of Testmat and Their Composition

The calling tree for this program is



The Matrix Multiply phase is augmented to employ a Jog&Jump-Outer-In Policy as presented in the conclusion of Chapter 5. The remainder of the augmented program is shown below.

f	0.1	0.2	0.3	0.4	0.5
$A = 50$	20 (48)	21 (43)	**22** (46)	16 (44)	12 (42)
$A = 100$	37 (94)	42 (86)	**44** (95)	30 (90)	30 (98)
$A = 200$	77 (162)	85 (176)	**86** (185)	60 (175)	45 (143)
$A = 400$	**151** (365)	150 (309)	150 (315)	111 (254)	86 (193)
$A = 800$	**239** (704)	233 (604)	229 (548)	189 (492)	185 (434)
$A = 1200$	**286** (945)	249 (858)	242 (760)	239 (710)	233 (604)
$A = 1600$	**289** (984)	288 (963)	287 (951)	249 (858)	242 (760)

Figure 7.6: $P_{ave}(A_{max})$ Versus Fraction Allocated to Smaller Loop Nest for Several Resource Limits in Testmat.

```
def testmat n R f =
  {frames = R - 1;
   rmat_frames = floor (frames * f);
   mm_frames = frames - rmat_frames;
   A = rmat n rmat_frames;
   in MatrixMult' A A n n n mm_frames};

DEF rmat n frames =
  {A = matrix ((1,n),(1,n));
   {FOR i <- 1 TO n bound (max 2 (floor frames / 3)) DO
       {FOR j <- 1 to n bound 2 DO
          A[i,j] = 1.0}};
    in A};
```

The behavior of this program as a function of A and f is shown in Figure 7.6. In this case, $f = 0.3$ offers maximal P_{ave} until resources are relaxed beyond $A = 200$ point. From $A = 400$ on, the smallest fraction is best.

The threshold policy for the producer is easily incorporated into the resource splitting code as follows.

```
def testmat n R f =
  {frames = R - 1;
   rmat_frames = min (3*n) (floor (frames * f));
   mm_frames = frames - rmat_frames;
   A = rmatx n rmat_frames;
   in MatrixMult' A A n n n mm_frames};
```

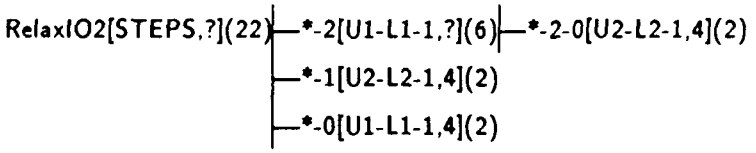
f	0.05	0.1	0.2	0.3	0.4
$A = 50$	20 (48)	20 (48)	21 (43)	**22** (46)	16 (44)
$A = 100$	34 (91)	37 (94)	42 (86)	**44** (95)	30 (90)
$A = 200$	62 (153)	77 (162)	**85** (176)	**85** (184)	**85** (184)
$A = 400$	128 (354)	151 (365)	**154** (365)	**154** (365)	**154** (365)
$A = 800$	230 (713)	**239** (700)	**239** (700)	**239** (700)	**239** (700)
$A = 1200$	**287** (941)	**287** (941)	**287** (941)	**287** (941)	**287** (941)
$A = 1600$	**289** (981)	**289** (981)	**289** (981)	**289** (981)	**289** (981)

Figure 7.7: $P_{ave}(A_{max})$ Versus *Limited* Fraction Allocated to Smaller Loop Nest for Several Resource Limits in Testmat.

The behavior with this policy is shown in Figure 7.7. In this case, $f = 0.3$ is optimal throughout.

7.3.3 Boundary conditions: relax

We now consider a more complex example, where branching arises because of loops that handle boundary conditions. Rather than explore the range of possible resource partitions, we implement the threshold policy and inspect its behavior. The program we study is the version of `Relax` discussed in Chapter 2, where the boundary conditions are implemented as independent loops. The calling tree for this is as follows.



The top-level loop is an Unfold-If-Necessary (UNIN) loop that invokes three subordinate loops. Two of these, the boundary loops, have $\mu = 4$. The third is an Unfold-If-Possible (UIP) loop, that invokes a subordinate loop with $\mu = 4$. The R_{min} proportions at the branch are 2/10, 2/10, and 6/10, respectively. However, the two loops with bounded unfolding have a

small R_{max} threshold, $2 \leq A \leq 4$. This is important both for implementing a threshold policy and for determining the unfolding of the outermost loop. The k-bound for the outermost loop, k_o, is computed by assuming all the loops within it unfold to their maximum useful extent, as can be seen in the augmented *Id* program below. This determines the resources available at the branch point. The k-bounds of the two small loops are computed using the R_{min} proportions and the useful unfolding limit. The remainder of the resources are devoted to the **interior** loop nest, as dictated by the threshold splitting policy. The k-bound of the middle loop is computed by assuming the innermost loop has minimum unfolding. Finally, this determines the resources available to the innermost loop, and hence its unfolding. The top level block, `RelaxIO`, is augmented with code to determine the k-bounds, while the loops are augmented with new arguments, the k-bound for each level.

```
%%% Frame-limited relax with inner-out policy and threshold splitting
def RelaxIO2 A steps frames =
  {((l1,u1),(l2,u2)) = (2D_bounds A);
   n = u1 - l1 - 1;                              % iterations for interior
   ko = max 2 (min steps (floor (frames / (5*n - 1)))); % assume max unfolding within
   F = floor (frames / ko) - 1;                  % available frames per iteration
   k_sides = max 2 (min 4 (floor (2/10)*F));     % resources to SIDES with threshold
   k_top   = max 2 (min 4 (floor (2/10)*F));     % resources to TOP with threshold
   F2 = F - k_sides - k_top;                     % remaining resources for INTERIOR
   k_int = max 2 (min n (floor (F2/3)));         % k_int assuming min unfolding within
   k_int_0 = max 2 (min 4 (floor (F2/k_int) - 1));      % k_int_0 get what is left
   in {for i <- 1 to steps bound ko do
         next A = relax_step A k_sides k_top k_int k_int_0;
       finally A}};
```

Here we have augmented the program as dictated by the threshold branch policy, with the loop nest policy developed in previous chapters applied to each branch. We have not tried to retain the test parameter, f, as it is not clear how to arrive at a meaningful, and yet relatively small, space of alternatives when multiway branching occurs within a loop. Thus, in our experimental methodology we take another step back from optimality concerns and try to establish reasonable "figures of merit" to judge our solution. Recall, our goal is to take a given collection of resources and make the the best possible use of them. Thus, one criterion is that the peak resource load should be close to the total available. The second column of Figure 7.8 shows the peak activation frame usage of our augmented program measured under the ideal execution model at several resource levels. We do not expect that this will equal the total resource, shown in the first column, since the ideal model may not represent the worst-case

A	A_{max}	A_{ave}	σ	P_{ave}
25	21	15.70	3.10	22
50	45	28.86	13.72	48
100	87	50.31	22.56	89
200	166	74.10	35.28	114
300	242	121.37	61.24	117
400	295	168.12	94.58	114
600	430	218.17	143.38	114
800	546	259.78	175.80	117
1,000	546	259.78	175.80	117

Figure 7.8: Behavior of RelaxIO2, 10 steps on a 15×15 Mesh, for Several Activation Frame Limits

execution schedule. Here we see a peak usage of at least 50% of the total available resources in every case.

A second criterion is that the resource load be fairly even, especially when resources are tightly constrained. If the load were perfectly uniform, then the mean would be equal to the maximum, whereas if the resource profile were to have a very large spike, the ratio A_{ave}/A_{max} would tend toward zero. Thus, we will use this ratio as a measure of the smoothness of the resource load. The third column of Figure 7.8 shows the mean activation frame usage. It is generally about half the peak usage, although under tight resource constraints it is a little higher. The fourth column gives the standard deviation, to further characterize the load.

The final criterion is that the exposed parallelism should increase nearly linearly with the available resources, within the range in which it is possible to make effective use of resources. Here we see that the increase is quite good up to some point between 100 and 200 frames. Beyond this there is almost no return on additional resources. Parallelism of order 100 on such a small problem is quite good, and it is interesting how sharply the improvement falls off.

As a slight variation on this experiment, Figure 7.9 shows the behavior of this program, where the unfolding of the innermost loops is kept at 2, rather than allowed to reach $\mu = 4$. The effect of this change is quite small. First, consider the two loops that compute the boundary. Under tight resource constraints these are controlled by the R_{min} limits, so it does not matter what is the maximum unfolding. Under loose constraints, the difference between R_{min} and R_{max} is so small for these loops that the change in resource levels has little effect on either

A	A_{max}	A_{ave}	σ	P_{ave}
25	21	15.70	3.10	22
50	45	28.86	13.72	48
100	87	50.31	22.56	89
200	188	96.33	52.35	90
300	272	142.85	85.34	99
400	336	159.52	109.79	90
600	376	190.48	121.97	109
800	376	190.48	121.97	109
1,000	376	190.48	121.97	109

Figure 7.9: Behavior of Relax_IO2, 10 steps on a 15 × 15 Mesh, for Several Activation Frame Limits With Innermost Loops Tightly Bound

A	A_{max}	A_{ave}	σ	P_{ave}
25	21	15.71	3.08	22
50	45	28.92	13.69	48
100	87	50.40	22.16	90
200	166	74.35	35.39	114
300	242	121.72	61.37	117
400	242	121.72	61.37	117
600	242	121.72	61.37	117
800	242	121.72	61.37	117
1,000	242	121.72	61.37	117

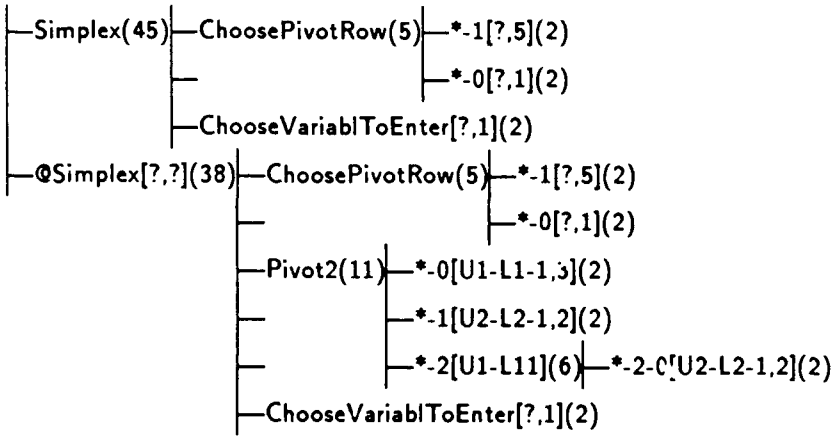
Figure 7.10: Behavior of Relax_IO2, 10 steps on a 15 × 15 Mesh, for Several Activation Frame Limits, with Structure Storage Limiting Outer Loop Unfolding to 4.

these or the main loop nest. In the main nest, the middle loop is given priority; thus, the innermost is only allowed to unroll when resources are plentiful. This is already in the region where the parallelism curve flattens.

To complete this study, we consider the additional constraints imposed by structure storage. Since structure allocation is tied to the outer loop, independent structure limits simply place an upper bound on k_o. The effect of this is shown in Figure 7.10, where $k_o \leq 4$. Since the outer loop unfolds only if necessary, this fixes the configuration after a certain resource point.

7.3.4 Transform and test: simplex

As a final example we consider an implementation of the Simplex algorithm using Bland's anticycling rule[73]. It exhibits a more complex calling tree, conditionals, and loops where the iteration count is not a simple function of the problem size. The calling tree is shown below. Recall, at each loop node the two quantities in brackets represent a symbolic expression for the iteration count and the maximum useful unfolding, respectively. A question mark indicates that the corresponding item could not be determined.



The first branch arises because the initial pivot row and column are determined outside the loop. Within `choose_pivot_row` are two loops. Inspection of the program shows that the first locates the first candidate row and the second finds the best row. While they execute nearly serially, our compile-time analysis is not powerful enough to discover this. Both have unknown iteration counts, but limited useful unfolding (LUU), as determined through path-length analysis. The `ChooseVariableToEnter` also has LUU, so this side of the top level branch has activation frame usage that ranges from 7 to 10. The threshold splitting policy takes care of it nicely and is applied recursively.

The other side of the top level branch is the main loop of the Simplex algorithm. It is marked as Unfold-If-Necessary (UNIN), because a new matrix is allocated in each iteration. Within the loop we have tree branches; two are as discussed above and the remaining is the pivot step. It contains three loops, two have LUU and the last, Pivot2-2, is a nested loop. This is the one loop in the entire tree marked as a primary target for unfolding. The loop it contains, the row elimination, also has LUU.

By applying the threshold splitting rule recursively and allowing the UNIN loop to unfold only when all other loops are unfolded to the maximum useful extent, attention and resources are focused on the one loop that really matters — the elimination step of the pivot. With more sophisticated analysis we could improve the policy only slightly. For example, we could recognize that the three loops involved in locating the pivot row and column are essentially serial, but this only allows an additional eight activation frames to be devoted to the main loop.

7.4 Conclusions

Adjacent loops require that a decision be made as to how resources are split between the various branches. The potential complexity in the relationships between such loops appears to be quite troublesome, but a simple policy based on size, rather than relationship, performs quite well. We can determine the proportion of resources allocated to each branch with certainty when all the loops are tightly bounded. For a range of fairly tight resource constraints, this proportion behaves reasonably well. In many cases, it is crucial to determine the maximum resources that a branch can use, so that additional resources can be diverted to the other, more complex branches, once the branch is completely unfolded. Otherwise, if the branches represent computations of roughly equal complexity, we do well by giving them roughly equal resources.

Chapter 8

A Case Study

In this chapter we draw the developments of the preceding chapters together into a coherent policy and carry out this policy on a sizable program in order to evaluate its effectiveness "in the large". The program we examine is a hydrodynamics and heat conduction simulation, known as the Simple Code[25], which has been used as an application benchmark for many parallel computers. Parallelism and resource profiles for this program appear in Chapter 2. The calling tree for this program is large, containing roughly 300 nodes. Compile-time analysis focuses attention on a fairly small collection of loops, representing the primary computational and resource complexity of the program. Policy guidelines developed previously for handling branching, structure recurrences, and nesting are shown to perform quite well.

8.1 Policy Overview

In this section we bring together the guidelines developed for handling particular aspects of program structure. This is presented in terms of the process for augmenting a loop program with resource management code. We begin with an overview of the kinds of analysis required and then discuss how the program is augmented with resource management code.

8.1.1 Analysis

Build the Calling Tree:

The backbone for our policy formulation is the call tree, and hence a representation of this will need to be constructed. This tree is then annotated with various kinds of information. Note, the requirement that it be possible to construct the calling tree restricts OUR work to

a restricted class of programs, as it excludes general recursion. We discuss this restriction further in the final chapter. Here, we comment on two other subtle aspects of this analysis. We have scarcely addressed the issue of conditional execution, but where conditionals appear in other than the leaves of the tree they need to be represented in the tree itself, as they affect the resource analysis and the parallelism estimates that underlie the policy guidelines. Our experience with scientific programs expressed in *Id* suggests that the occurrence of high-level conditionals is fairly rare, but the set of examples available to date is far too small to make a strong statement to that effect. Our case study employs conditionals, but well down inside the tree.

The more subtle issue is the role of higher-order functions. It is possible that the program has a simple call tree, but constructing it may be difficult as it involves detecting how higher-order functions are used. Our case study makes extensive use of higher-order functions in the abstractions it builds, but through a series of compiler optimizations (tuple-elimination, arity analysis, and in-lining)[85] the "higher-orderness" is compiled away to yield straightforward loop code[8, 11].

Identify loops with Limited Useful Unfolding:

The next important analysis step is to detect loops with limited useful unfolding. This is crucial because allowing these loops to unfold beyond this amount squanders resources. Usually innermost loops will have a fairly small useful unfolding and, because they are also deeply nested, any resources allotted to them are multiplied manyfold. Using the simple path-length techniques of Chapter 4, the useful unfolding of innermost loops is easily estimated. That analysis only takes account of dependencies represented explicitly as arcs in the dataflow graph, so it may overestimate the useful unfolding of a loop with a dependence via a data structure. On the other hand, it assumes that communication is instantaneous and that *I-fetches* are serviced immediately, rather than being deferred. Both of these factors tend to increase the unfolding of the loop in practice, although not necessarily the amount of parallelism. We discuss this issue further in the next chapter.

More powerful dependence analysis is required to detect other kinds of LUU loops. The usual subscript analysis techniques[88] can be used to detect dependencies across iterations through arrays, and techniques similar to Doacross scheduling[28] can be used to translate this into useful unfolding. More unusual analysis is required to understand the dynamics of structure

recurrences. For example, the relaxation program studied in Section 5.3.1 demonstrated limited unfolding along the structure recurrence, for a given sized mesh. To estimate this we would need to consider the length of the point-to-point dependence between values in the mesh as compared to the number of steps taken in filling a mesh. Finally, there is an important class of LUU loops that can only be detected through deep dependence analysis. Basically, we need to determine that essentially an entire data structure is reduced to a particular scalar value which is broadcast to essentially all the elements of the next version of the structure, as pictured in Figure 8.1. Detecting this situation may be possible through analysis similar to that for static range checking[49, 68]. It frequently arises in testing for convergence, as in the relaxation example shown in Figure 2.9, but may involve dependencies through several data structures, computed in completely different parts of the program, as in the Simple code. The experiments presented here suggest that fairly good results can be obtained with almost no symbolic dependence analysis.

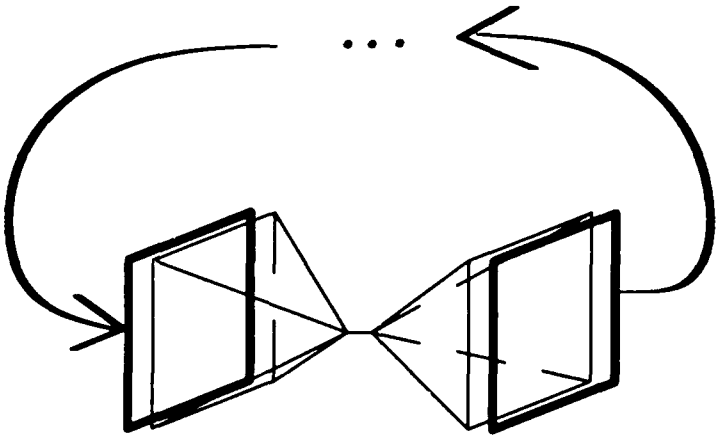


Figure 8.1: Common Scenario with Reduction and Broadcast Causing Limiting Useful Unfolding

Analysis of iteration counts is important for several reasons. It plays a role in detecting LUU loops, because through simplifying the start, step, and end expressions we may discover that the number of iterations is a small constant. This often arises through building abstractions for regions of a n-dimensional range, *e.g.*, boundary and interior. The programmer never sees the loop explicitly and so does not eliminate trivial loops as one would if the program were expressed in a lower-level programming style. More generally, where the iteration count can be expressed as a simple function of program variables, we have a way of dynamically determining

the maximum k-bound. We try to lift this point of definition as high in the call-tree as possible, in order to make the most use of this information in formulating the policy. Iteration counts are key in determining whether an enclosing UNIN loop should unfold and in determining the resource threshold at the enclosing branch.

Identify Unfold-only-If-Necessary loops:

The second category of loops we need to recognize are those which may offer arbitrary unfolding, but have very large resource costs. Usually a loop that contains an allocation of a large data structure, perhaps several levels down in the calling tree, will fall into this category. Interestingly, a loop with a structure recurrence that has limited useful unfolding due to a deep scalar dependence will usually be treated as an UNIN loop if it escapes dependence analysis, however, this will still allow the loop-bounding policy to perform effectively. The loop will only be allowed to unfold if resources are plentiful. Loops that are not placed in either of these two categories will be treated as prime candidates for unfolding.

Structure storage analysis

In addition to detecting these structural characteristics, we need to estimate the resource requirements of subtrees in terms of their k-parameters. This involves analysis of both I-structure storage and local storage.

Above we discussed detecting allocation patterns for large data structures. To properly k-bound structure recurrences requires locating the loop in which the recurrence is present and forming a symbolic expression of the size of the data structure in terms of the free variables of the loop. Recall, in order to recycle data structures as discussed in Chapter 3, the size and shape of the structure must not change from one iteration to the next. Determining the order of the recurrence is tantamount to detecting the lifetime of the structure variable. Analytical techniques along these lines are under development at MIT[89] based on flow analysis[57, 70]. Where the lifetime is defined by a window of iterations of a loop, but the structure changes size or shape, k-bounding still controls the number of concurrent versions of the structure, although more expensive storage management techniques are required. To meet a storage requirement guarantee, we need to be able to estimate the maximum size of a version of the structure. It is rare to have multiple levels of structure recurrences, but it does occur in real applications. In this case, the structure storage requirement is linear in multiple k-bounds, much like the expressions for local storage discussed below.

Local storage analysis

Analysis of local storage essentially involves walking up the calling tree forming a polynomial in the k-bounds, representing the size of the active portion of the execution tree. If the storage unit is an activation frame, this calculation is as follows. The requirement of an acyclic code-block is one larger than the sum of the requirements of its children. For a loop code-block, we do the same calculation and multiply by the k-bound for the cyclic portion, and then add in the children in the loop prologue and epilogue. If the storage unit is a token, similar rules apply, but the coefficient used for a code-block is determined by solving the linear program extracted from the graph, as in Appendix B.

For both forms of local storage, the algorithm we have described gives the worst-case requirement assuming all code-blocks execute in parallel. This is nearly the case for graphs compiled from *Id*, because the activation frame or context may be allocated before any of the arguments to a code-block are even computed. As we saw in Chapter 2, much of the program unfolds very rapidly. However, where it is possible to determine that two activations cannot overlap, the summation is replaced by a maximum. Thus, in general, finding a tight bound on the maximum resource requirement of a calling tree involves solving a constraint system, much as for the token storage within a code-block. For a static set of k-bounds this can be done precisely as for tokens, but solving a symbolic system is tricky. For the kind of structures that occur in practice, one expects that simple heuristics will suffice. We can start out with the worst-case expression, assuming all the activations are concurrent, and where we can prove that subtree A precedes subtree B and the resource expression for one dominates that for the other, we drop the resource term for the smaller in the summation appearing in a common parent. Conditionals introduce a similar complication, but we can begin by assuming the resource requirement is the sum of the two arms and improve the estimate through a branch-and-bound technique.

Even ignoring the issue of improving the estimate by recognizing dependencies, these resource expressions can get quite complicated. However, we can implement an effective policy with a few of simple pieces of information. For each node, we compute the number of frames required by the subtree rooted at the node with all k-bounds set to their minimum value, usually 2. This provides the coefficients for the R_{min} ratio policy at the branches. Where possible, we build a symbolic resource expression for loop nests, where the bound for each loop is the

maximum useful unfolding, but place a limit on the complexity of the expression, *e.g.*, a linear expression in a single program variable. This is used for threshold splitting. Also, for simple loop nests we need to record the resources per unit k for each loop level. Using these resource coefficients, it is straightforward to generate code to solve for each of the k-bounds one at a time, as in Chapter 5.

8.1.2 Program augmentation

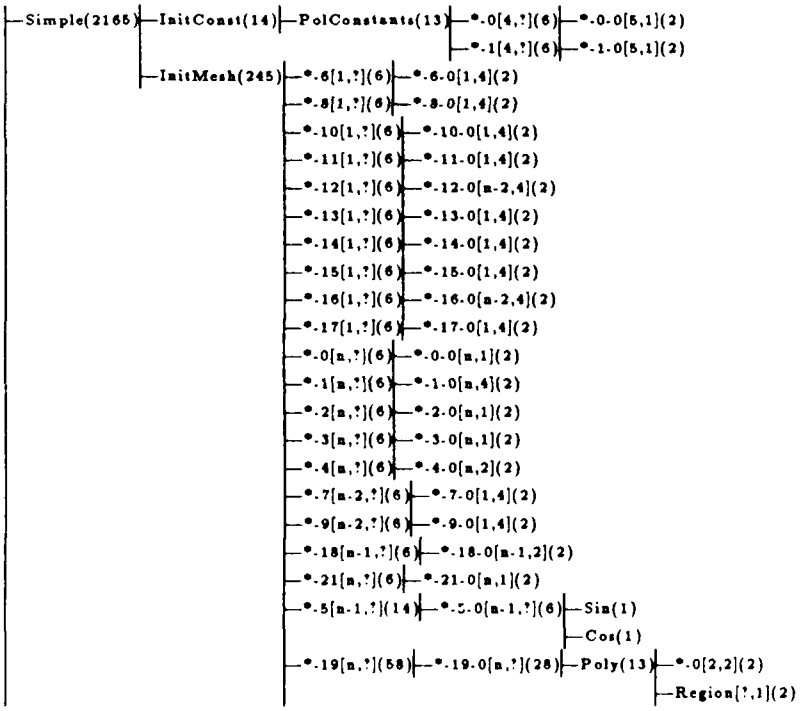
Having analyzed the program from the leaves of the calling tree up to the root, we augment it with resource management code working from the root down. To simplify the discussion, we focus on local storage. At each node we have an R_{min} value, possibly an R_{max} expression, possibly an expression for the maximum useful k-bound, and a resource coefficient. At the root we introduce a new parameter, r, representing the total resources available. At branch points we introduce code to divide the resources coming down to the branch between the subtrees at the branch. At loops, we introduce code to solve for k in terms of the incoming resource parameter. In many nested loops, we will be able to solve for all the k-bounds of the nest at the level of the outer loop, so the resource management code within the nest simply passes in the k-bounds. In other cases, we must settle for an Outer-In policy, solving for the k-bound at each level assuming the inner loops are tightly constrained and passing the available resources into each loop to solve for the next k-bound. In the next sections we illustrate this approach on a full-scale example.

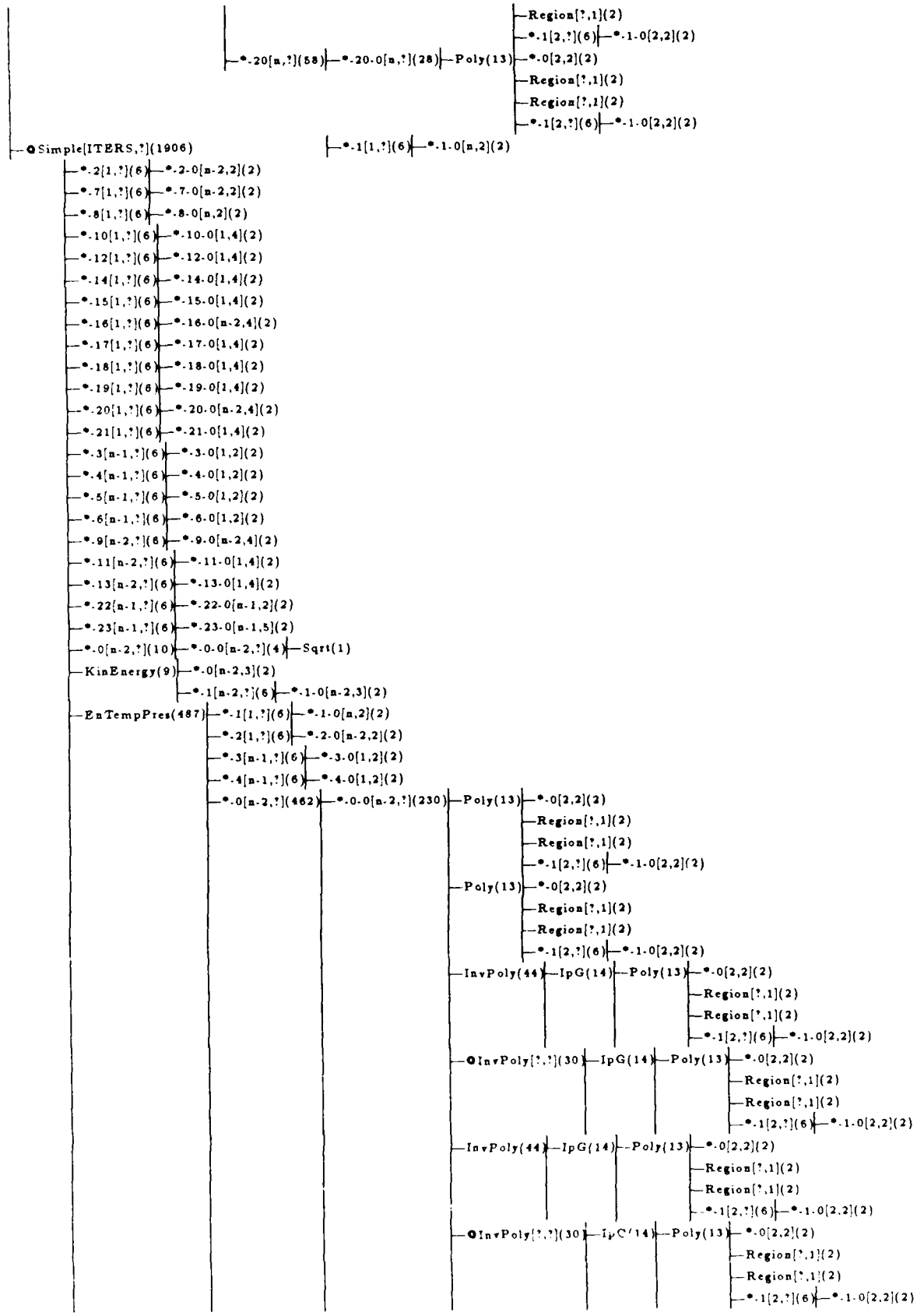
8.2 Simple — the Analysis Phase

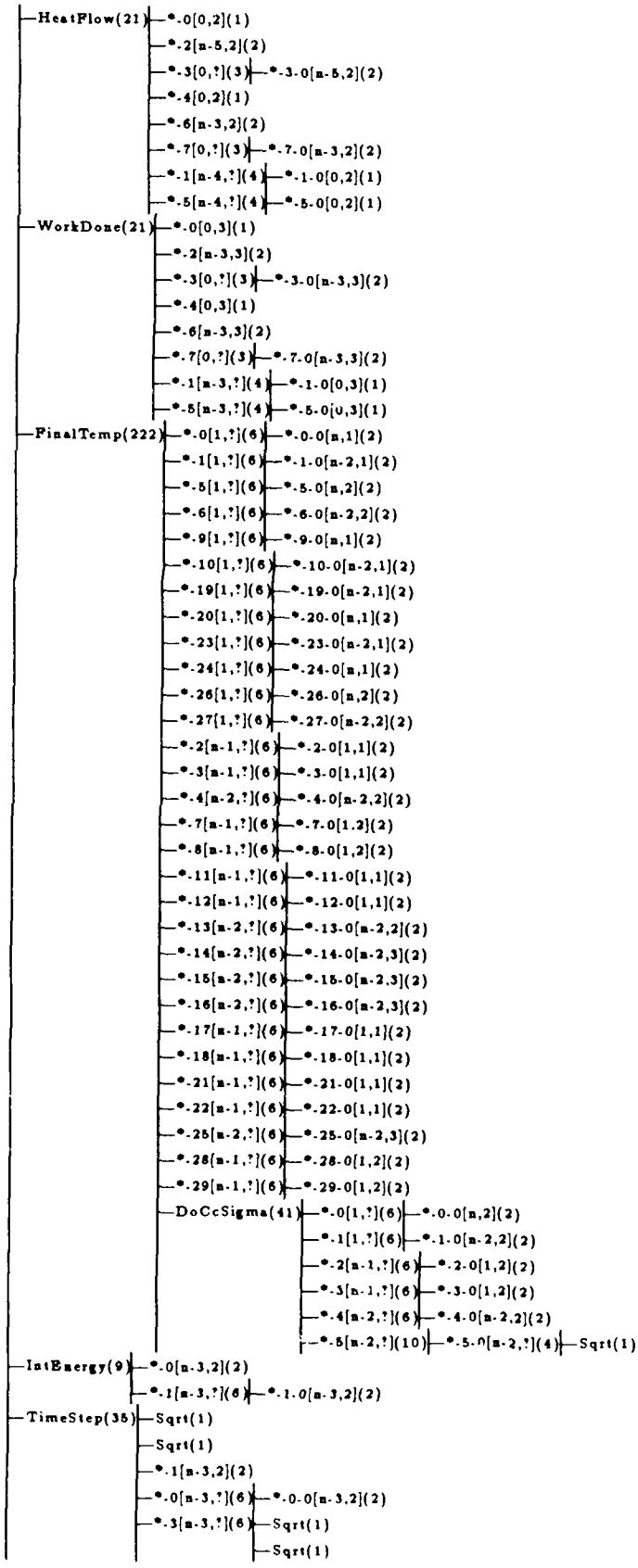
The program we examine is the Simple Code, which simulates the behavior of fluid in a sphere using a Lagrangian formulation. To simplify the problem, only a semi-circular cross-sectional area is considered. The program does not vectorize easily, due to extensive table lookups inside key inner loops. The particular version we use was developed to demonstrate the virtues of programming scientific applications at a very abstract level, using higher-order functions and I-structures[11]. A declarative style is adopted, in which node or zone formulae are applied to generalized coordinate functions over finite ranges. The resulting program directly reflects the equations describing the physics of the problem. This is quite far from the versions of

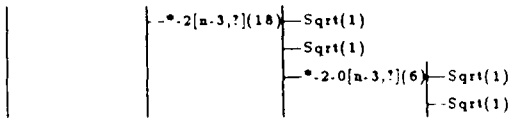
this program studied elsewhere[65, 78], which employ a very low-level programming methodology. Through some powerful compilation techniques a fairly efficient executable program is produced[11]. However, the final program is a huge complex of nested loops, with little correlation to the original text. The resource behavior of the unconstrained program is terrible, as can be seen in Figure 2.11 of Chapter 2. Below, we go through the steps involved in analyzing this program and augmenting it to produce a well-behaved bounded-loop program. Finally, we show how well it performs.

The first step in our analysis is to construct the calling tree. The complete tree is shown below. It contains a total of 292 nodes, including 257 loops, with a maximum loop nesting of six. Thus, finding an optimal assignment of the k-bounds is out of the question. The call tree exhibits a host of different constructs: branching, nesting, structure recurrences, reductions, conditionals, and while loops. Where a loop has code-block invocations both within the body of the loop and outside of the body, it is represented as a pair of nodes at the same level in the tree with the body indicated by an ampersand. Rather than perform detailed analysis of conditionals, we have adopted a conservative approach and charged for both branches. The primary effect of this is that resources are allocated for the **Region** loop within **Poly**, even though it is enclosed in a conditional. Conditionals appear within expressions elsewhere in the program, but do not effect activation frame estimates.




*-20[n,?](58)
*-20-0[n,?](28)
Poly(13)
Region[?,1](2)
*-1[2,?](6)
*-1-0[2,2](2)
*-0[2,2](2)
Region[?,1](2)
Region[?,1](2)
*-1[2,?](6)
*-1-0[2,2](2)
*-1[1,?](6)
*-1-0[n,2](2)
QSimple[ITERS,?](1906)
*-2[1,?](6)
*-2-0[n-2,2](2)
*-7[1,?](6)
*-7-0[n-2,2](2)
*-8[1,?](6)
*-8-0[n,2](2)
*-10[1,?](6)
*-10-0[1,4](2)
*-12[1,?](6)
*-12-0[1,4](2)
*-14[1,?](6)
*-14-0[1,4](2)
*-15[1,?](6)
*-15-0[1,4](2)
*-16[1,?](6)
*-16-0[n-2,4](2)
*-17[1,?](6)
*-17-0[1,4](2)
*-18[1,?](6)
*-18-0[1,4](2)
*-19[1,?](6)
*-19-0[1,4](2)
*-20[1,?](6)
*-20-0[n-2,4](2)
*-21[1,?](6)
*-21-0[1,4](2)
*-3[n-1,?](6)
*-3-0[1,2](2)
*-4[n-1,?](6)
*-4-0[1,2](2)
*-5[n-1,?](6)
*-5-0[1,2](2)
*-6[n-1,?](6)
*-6-0[1,2](2)
*-9[n-2,?](6)
*-9-0[n-2,4](2)
*-11[n-2,?](6)
*-11-0[1,4](2)
*-13[n-2,?](6)
*-13-0[1,4](2)
*-22[n-1,?](6)
*-22-0[n-1,2](2)
*-23[n-1,?](6)
*-23-0[n-1,5](2)
*-0[n-2,?](10)
*-0-0[n-2,?](4)
Sqrt(1)
KinEnergy(9)
*-0[n-2,3](2)
*-1[n-2,?](6)
*-1-0[n-2,3](2)
EnTempPres(487)
*-1[1,?](6)
*-1-0[n,2](2)
*-2[1,?](6)
*-2-0[n-2,2](2)
*-3[n-1,?](6)
*-3-0[1,2](2)
*-4[n-1,?](6)
*-4-0[1,2](2)
*-0[n-2,?](462)
*-0-0[n-2,?](230)
Poly(13)
*-0[2,2](2)
Region[?,1](2)
Region[?,1](2)
*-1[2,?](6)
*-1-0[2,2](2)
Poly(13)
*-0[2,2](2)
Region[?,1](2)
Region[?,1](2)
*-1[2,?](6)
*-1-0[2,2](2)
InvPoly(44)
IpG(14)
Poly(13)
*-0[2,2](2)
Region[?,1](2)
Region[?,1](2)
*-1[2,?](6)
*-1-0[2,2](2)
QInvPoly[?,?](30)
IpG(14)
Poly(13)
*-0[2,2](2)
Region[?,1](2)
Region[?,1](2)
*-1[2,?](6)
*-1-0[2,2](2)
InvPoly(44)
IpG(14)
Poly(13)
*-0[2,2](2)
Region[?,1](2)
Region[?,1](2)
*-1[2,?](6)
*-1-0[2,2](2)
QInvPoly[?,?](30)
IpG(14)
Poly(13)
*-0[2,2](2)
Region[?,1](2)
Region[?,1](2)
*-1[2,?](6)
*-1-0[2,2](2)


HeatFlow(21)
*-0[0,2](1)
*-2[n-5,2](2)
*-3[0,?](3)
*-3-0[n-5,2](2)
*-4[0,2](1)
*-6[n-3,2](2)
*-7[0,?](3)
*-7-0[n-3,2](2)
*-1[n-4,?](4)
*-1-0[0,2](1)
*-5[n-4,?](4)
*-5-0[0,2](1)
WorkDone(21)
*-0[0,3](1)
*-2[n-3,3](2)
*-3[0,?](3)
*-3-0[n-3,3](2)
*-4[0,3](1)
*-6[n-3,3](2)
*-7[0,?](3)
*-7-0[n-3,3](2)
*-1[n-3,?](4)
*-1-0[0,3](1)
*-5[n-3,?](4)
*-5-0[0,3](1)
FinalTemp(222)
*-0[1,?](6)
*-0-0[n,1](2)
*-1[1,?](6)
*-1-0[n-2,1](2)
*-5[1,?](6)
*-5-0[n,2](2)
*-6[1,?](6)
*-6-0[n-2,2](2)
*-9[1,?](6)
*-9-0[n,1](2)
*-10[1,?](6)
*-10-0[n-2,1](2)
*-19[1,?](6)
*-19-0[n-2,1](2)
*-20[1,?](6)
*-20-0[n,1](2)
*-23[1,?](6)
*-23-0[n-2,1](2)
*-24[1,?](6)
*-24-0[n,1](2)
*-26[1,?](6)
*-26-0[n,2](2)
*-27[1,?](6)
*-27-0[n-2,2](2)
*-2[n-1,?](6)
*-2-0[1,1](2)
*-3[n-1,?](6)
*-3-0[1,1](2)
*-4[n-2,?](6)
*-4-0[n-2,2](2)
*-7[n-1,?](6)
*-7-0[1,2](2)
*-8[n-1,?](6)
*-8-0[1,2](2)
*-11[n-1,?](6)
*-11-0[1,1](2)
*-12[n-1,?](6)
*-12-0[1,1](2)
*-13[n-2,?](6)
*-13-0[n-2,2](2)
*-14[n-2,?](6)
*-14-0[n-2,3](2)
*-15[n-2,?](6)
*-15-0[n-2,3](2)
*-16[n-2,?](6)
*-16-0[n-2,3](2)
*-17[n-1,?](6)
*-17-0[1,1](2)
*-18[n-1,?](6)
*-18-0[1,1](2)
*-21[n-1,?](6)
*-21-0[1,1](2)
*-22[n-1,?](6)
*-22-0[1,1](2)
*-25[n-2,?](6)
*-25-0[n-2,3](2)
*-28[n-1,?](6)
*-28-0[1,2](2)
*-29[n-1,?](6)
*-29-0[1,2](2)
DoCcSigma(41)
*-0[1,?](6)
*-0-0[n,2](2)
*-1[1,?](6)
*-1-0[n-2,2](2)
*-2[n-1,?](6)
*-2-0[1,2](2)
*-3[n-1,?](6)
*-3-0[1,2](2)
*-4[n-2,?](6)
*-4-0[n-2,2](2)
*-5[n-2,?](10)
*-5-0[n-2,?](4)
Sqrt(1)
IntEnergy(9)
*-0[n-3,2](2)
*-1[n-3,?](6)
*-1-0[n-3,2](2)
TimeStep(35)
Sqrt(1)
Sqrt(1)
*-1[n-3,2](2)
*-0[n-3,?](6)
*-0-0[n-3,2](2)
*-3[n-3,?](6)
Sqrt(1)
Sqrt(1)



Secondly, we have performed loop iteration count analysis by propagating the start, end, and step expressions of FOR loops backward through the graph, building a symbolic expression for the iteration count at the highest possible point in the tree. We then simplify the expression to put it in canonical form. In this program this analysis is very successful, picking up simple linear expressions for all the FOR loops. The loops where no iteration count is determined are indicated by "?", *e.g.*, **InvPoly** and **Region**. These are, indeed, WHILE loops. **Region** performs a table lookup and **InvPoly** uses this function repeatedly to invert a piecewise linear function. Many of the FOR loops execute a small, fixed number of iterations. This arises as a result of the abstraction for applying a collection of functions to rectangular regions of a 2-dimensional range. In many cases the range is just a vector or a point. The other source of small, fixed iteration count loops is the abstraction for evaluating polynomials based on a table of coefficients.

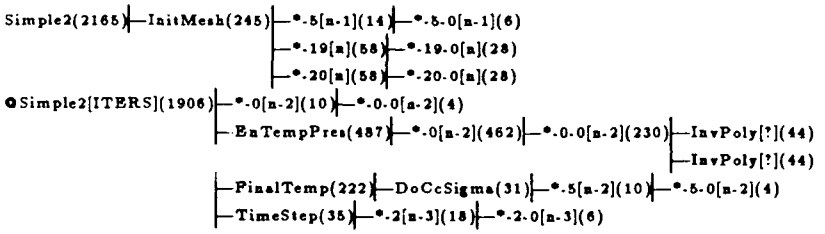
Thirdly, we have performed simple loop unfolding analysis on innermost loops containing no invocations. This is straightforward path-length analysis. No subscript analysis is performed, so we may overestimate certain loops — two pairs of loops inside of **FinalTemp** implement a forward and back solver with such recurrences. Where applicable, the results of this analysis appear as the second expression in square brackets, otherwise a "?" appears. This analysis identifies one of the loops with unknown iteration count, **Region**, as having limited useful unfolding. The other WHILE loop, **InvPoly**, also has limited unfolding, but interprocedural dependence analysis is required to identify it.

From the viewpoint of loop bounding policy, we can essentially ignore these loops with limited useful unfolding, as they enter primarily in the resource splitting phase. In this program, the loops with bounded natural unfolding have $\mu \leq 5$, so for the purposes of this study we will fix them at 2 and consider them as we do acyclic blocks. This affects the experiment only at fairly high resource levels. Each of these two analysis techniques in isolation reduces the size of tree we need to deal with to 143 nodes, or roughly in half. Applied together, *i.e.*, eliminating all LUV nodes that contain only LUV nodes, the tree is reduced to 75 nodes.

8.3 Program Augmentation Phase

The first-order policy decision is identifying the outermost loop as Unfold-Only-If-Necessary (UNIN). This is straightforward because the structure resource expression is $25n^2k_{simple}$. All the structure allocations are per outer loop iteration, although they appear nested somewhat in the tree. This program can be put in the form discussed in Chapter 3 by lifting these allocations. There is actually a scalar dependence that limits the unfolding of the outer loop to 2, but this is a very deep dependence and very powerful analysis techniques would be required to detect it.

Having reached this decision, the policy for the loop nests containing only one level of interesting unfolding is clear: simply divide by R_{min} to get the k-bound, once resources are split at the top level. The threshold for resource splitting takes effect at the point indicated by the iteration count analysis. Having eliminated these "single level" loops from consideration, the tree we need to examine more carefully is much smaller, as shown below:



More sophisticated analysis could take us a bit farther and simplify this nest in two ways. Path-length analysis across acyclic code-blocks in innermost loops would place a limit on the TimeStep branch, as the inner loop only includes calls to SQRT. Scalar dependence analysis across loops would show that the InvPoly nodes have bounded unfolding as well. Thus, we can see that compile-time analysis focuses our attention on the portion of the program that is essential, weeding out most of the insignificant structure. In the study that follows, we fix the unfolding of `InvPoly` at 2, but do not consider path analysis across code-block boundaries.

To see the effect of our policy, we have carried through the program augmentation process by hand. At each branch in the tree we group the children into classes corresponding to the number of interesting loop levels, *i.e.*, levels where unfolding can occur. The 0-level branches have a fixed resource requirement; all the loops appearing in these branches have a k-bound of 2. The 1-level loops require no policy to control and have a simple resource threshold. For

more complex loops, a simple Outer-In policy is followed. The branches of the static invocation tree shown above are ordered so that at a given branch level the loops increase in complexity down the page.

The implementation of the policy goes as follows. If a total of A activation frames are allotted to the entire program, 14 frames are directed to InitConst, $245A/(2165 - 14)$ frames are directed to the **InitMesh** branch, and the rest are given to the top-level loop. Within **InitMesh** there are 10 loops, actually these are doubly-nested loops with no useful unfolding. These are allotted 60 frames, as required for all k-bounds to be set to 2. Note that some of these loops only execute one iteration, and thus $k = 2$ is somewhat inappropriate. Unfortunately, the loop schema requires that the minimum number of frames are allocated anyway. These loops should have been "compiled away," in any case. In addition, **InitMesh** contains 9 loops with a single level of interesting unfolding, each contains an inner loop with limited unfolding. These are each allotted $6A/185$ frames, and the k-bounds are determined by dividing by 3. The collective maximum activation frame requirement of these 1-level loops is $27n$, so this is used as a threshold. Finally, **InitMesh** contains three complex loops, where an Outer-In policy is followed. The loop nest for **InitMesh-5** and **InitMesh-5-0**, containing calls to **Sin** and **Cos**, is allotted a fraction, $14/(14 + 58 + 58)$, of the resources remaining after subtracting the resources allotted to the 0-level and 1-level loops. The k-bounds are determined as in Chapter 5, and similarly a fraction, $58/(14 + 58 + 58)$, of the resources are allotted to each of the other nests.

Similarly, the branches in the body of Simple are grouped as 0-level loops, 1-level loops, 2-level loops, and then the individual code-blocks invoked within the loop. The R_{min} threshold splitting policy is employed recursively, and whenever we arrive at a loop nest with more than one level of useful unfolding an Outer-In policy is employed. This yields the results shown in Figure 8.2 for 4 steps on a 16×16 mesh and in Figure 8.3 for 4 steps on a 25×25 mesh.

Here we see that most of the potentially concurrent activations do indeed co-exist, as the measured peak activation frame usage, A_{max}, is consistently more than half the allotment, A. As more unfolding is permitted, however, this ratio drops. In comparison, the profile for Simple (4 iterations on a 16×16 mesh) shown in Chapter 2 has $A_{max} = 53,777$ and $P_{ave} = 1,322$. That profile was computed by allowing all the FOR loops to unfold to the total number of iterations, keeping the WHILE loops (**Region** and **InvPoly**) bounded at 2. The profile for this program shown in Chapter 3 achieves the same average parallelism with $A_{max} = 29,999$, by

A	A_{max}	A_{ave}	σ	P_{ave}
2,000	1,835	869	327	41
4,000	3,336	1,694	624	97
8,000	5,180	2,880	1,073	187
16,000	9,993	5,315	2,488	427
32,000	18,782	9,320	4,909	875

Figure 8.2: Behavior of k-Bounded Simple, 4 Iterations on a 16×16 Mesh for Several Activation Frame Limits and Structure Storage Limiting the Outer Loop to 2.

A	A_{max}	A_{ave}	σ	P_{ave}
2,000	1,903	913	339	39
4,000	3,541	1,684	687	78
8,000	6,682	3,192	1,361	156
16,000	10,810	5,915	2,274	408
32,000	20,094	11,041	5,133	1,020

Figure 8.3: Behavior of k-Bounded Simple, 4 Iterations on a 25×25 Mesh for Several Activation Frame Limits and Structure Storage Limiting the Outer Loop to 2.

setting the k-bound for the top-level loop to 2, as well. The smoothness ratio, A_{ave}/A_{max}, is roughly one-half throughout, and it actually improves slightly at the high end of the resource spectrum.

In the last column we observed that increasing the resource allotment allows more parallelism to be exposed. In fact, the parallelism increases more than linearly with resources, until we reach a point where most of the useful parallelism is exposed. One explanation for this is that as resources become more plentiful the partitioning of resources at the branch points more accurately reflects the relative sizes of the sub-computations. When resources are scarce, we are forced to give a sizable fraction to the smaller computation, but when they are more plentiful the thresholds "kick in" and the larger computation receives a comparatively larger share. In effect, the threshold policy establishes a piece-wise linear function representing the partitioning of resources at each branch point.

8.4 Conclusions

In bringing together the various policy components and applying the complete policy to a fairly large example, we are able to see how the analysis phases contribute to isolating the important portion of the program. Fairly straightforward analysis is surprisingly effective, although we have noted situations in which more powerful analysis techniques would pay off. Even though we are dealing with programs in the large, the information we need to extract is crude enough that the compilation task is tractable. The performance of the resulting policy is shown to be quite good on this program, under the ideal execution model.

Chapter 9

Conclusions and Evaluation

We have demonstrated that excessive resource usage is a serious hazard for general purpose, highly parallel computation; introduced a mechanism, k-bounded loops, for controlling parallelism and resource usage in programs; and developed a policy for employing the mechanism in the context of scientific programs. The mechanism is very attractive, as it provides tremendous control at a small execution cost and it makes the machine *easier* to build. Static analysis is used to estimate worst-case resource requirements and to form a crude estimate of the dynamic behavior of the program. Based on this analysis, we augment the program systematically with code to determine the k-bounds, based on top-level resource constraints and key program variables. The quality of the augmentation policies has been explored analytically and experimentally for simple program structures. Applying the technique to a large program, we find that the approach is quite adequate. With reasonable, although non-trivial, compilation techniques we can expect to assign k-bounds automatically on a fairly broad class of programs. A first-cut at the analysis was implemented in order to conduct the studies presented here, although the programs were augmented manually. In this final chapter, we adopt a more critical eye toward the work and examine some of its strengths and weaknesses, along with possibilities for future research.

9.1 Impact of the k-bounded Loop Mechanism

Occasionally, in a complex field of study a simple idea will promote a shift in perspective and thereby have far reaching impact. The k-bounded loop mechanism has had this role *vis a vis* dynamic dataflow architectures. It is efficient, in that few instructions are added to

the program, and leads to *simpler* architectures. In this section, we explore the impact of this idea on compilation to dataflow graphs and on the recent evolution of dynamic dataflow architectures.

9.1.1 What's in a tag

The impetus for developing the k-bounded loop mechanism was the excessive token storage requirements observed in executing even simple programs on tagged-token dataflow machines, often leading to deadlock. However, once the mechanism was understood, it became clear how to resolve a variety of nagging problems in tagged-token dataflow architectures. In particular, the structure of a tag diverged from that under the U-interpreter and became more meaningful for real machines.

In formulating linear constraint systems to model the token storage requirements of dataflow graphs, it became clear that for certain graphs there was no optimal solution, *i.e.*, without additional constraints the solution could be arbitrarily large. This occurs precisely when the loop has potentially unbounded unfolding. Furthermore, a loop has bounded unfolding if the graph forms a single strongly connected component[26]. By adding artificial dependence arcs to the graph, we can connect various components and cause the loop to have bounded unfolding, where the unfolding is no more than the number of loop variables. Rather than produce a graph with many dummy loop variables, we introduced a new tag manipulation operator and constructed a loop schema that allows the bound on loop unfolding to be set dynamically at the time of invocation.

This new loop schema places additional requirements on well-behaved graphs: when a token is produced on each output arc of a graph, all activity within the graph is complete. To meet this requirement, an otherwise well-behaved graph is embellished with signal arcs. It then becomes straightforward to tie completion of an iteration to initiation of the one logically k later. In doing so, iteration identifiers can be represented modulo k. Furthermore, completion of an entire code-block invocation, loop or acyclic, can be detected so that resources associated with the invocation can be released.

The significance of this is best appreciated with an understanding of the state-of-the-art in dynamic dataflow architectures when this mechanism was first proposed. In the formal model of tagged-token dataflow execution, the U-interpreter[14], a token identifies the instruction to

execute by static components of the tag: the code-block name, C, and the node within that code-block, S. The dynamic instance of that instruction is specified by a unique context identifier, U, and an iteration identifier relative to that context, I. Rules are given for generating new tags from old ones using only information local to the node. However, in order to accomplish this the tag space is used *sparsely*. The tag in a real machine must serve the same functions, but must be of fixed, reasonable, size.

In the MIT Tagged-Token Dataflow Architecture[9] the graph is stored in program memory, using an adjacency list representation. Thus, the code-block name, C, corresponds to the base address of the graph for a particular code-block and S is simply an offset from C. To allow multiple concurrent invocations of a code-block to share code, the token carries the number of a code-block register (CBR) which contains the base address for the code-block. The CBR number itself serves as a unique identifier for the invocation, replacing U. However, CBR numbers are drawn from a fixed pool, so context identifiers cannot be generated by a simple operation on input tags, as in the U-interpreter. They are explicitly allocated and released (at the graph level, not at the program level). This requires that it be possible to determine when an invocation is complete. Although various reference counting schemes are possible, this is addressed in the TTDA by embellishing the graphs with signal arcs, as discussed above.

The iteration identifier portion of the tag is also of fixed size on this machine. Therefore, tag manipulation operations in loops must check for overflow and, when it occurs, send the input token to an auxiliary destination. This causes a new CBR to be allocated, thereby acquiring a new collection of iteration identifiers. k-Bounded loops eliminate the need for overflow detection, by setting k no larger than what the iteration field can represent. The embellishments to the graph to detect completion of invocations are essentially what is required to detect completion of iterations, so by using this "completion signal" in an incremental fashion bounded loops can be implemented with little additional overhead.

The outcome of this line of development is that the nature of the tag has become quite different from that in the U-interpreter. Rather than uniquely identify each operation by its context in the overall execution of the program, the tag identifies the resources that are to be used in performing the operation. For example, instead of specifying the logical node C.S in the dataflow graph, <CBR,Offset> specifies the location in program memory that contains the instruction. However, the key for token matching, <CBR,Iteration,Offset> is still large,

requiring hashing or associative access to locate operands.

9.1.2 Eliminating matching

The logical evolution from this point is to manage the tag space carefully enough that it becomes possible to equate tags and addresses. In doing so, it is possible to avoid associative matching all together. Before we start paring down the tag space, we need to consider one additional aspect of the TTDA, loop *constant areas*. These are used to hold variables appearing within a loop, but constant over all iterations of the loop. The free variables of a loop have this role. A naive translation of loops to dataflow graphs will circulate these values through *L-switches*, even though they are unchanged by the loop. In the TTDA, the value of such loop constants are stored in a special data segment by the set-up portion of the loop. The inner product loop has five loop constants and only two loop variables. Without a constant area the number of instructions per iteration would double. In effect, the constant area is a one-level environment. The CBR contains the address of the associated constant area, as well as the address of the code-block. [1]

Thus, in the TTDA each code-block invocation has associated with it a code segment and a (constant) data segment, in addition to some dynamically determined amount of the waiting-matching store. The first step toward careful management of the tag space is to use the data segment to hold all the local data, *i.e.*, waiting tokens, for the loop. This requires a data segment per active iteration, call it an *activation frame*. To make this idea practical, we must deal with both the size of the frame and the number of frames. The simplest approach might be to assign a fixed amount of storage in the frame, a *slot*, for each node in the graph. Then, the frame can simply be addressed by the instruction offset. Unfortunately, this means that each slot is used only once and, at any time, only a small fraction of the slots will be in use. The utilization of storage will be quite poor, much as it is in static dataflow architectures[37]. By observing the data dependencies between nodes, slots can be reused to reduce the size of the frame. This requires that we separate the instruction offset and operand offset, so a portion of the tag specifies the location of the instruction to execute and a portion specifies the

[1] In handling loop set-up, iteration overflow, and assignment of iterations to processors, care must be taken that all iterations are associated with the proper constant area. Bounded loops simplify this problem since a constant area can be associated with each allocated iteration. By reusing the iteration identifier, the constant area is reused, regardless of how the loop iterations are distributed over the machine.

location of the operands. Assignment of slots in a frame is essentially like allocating registers via coloring[23]. With a single slot per dyadic node, two nodes *interfere* unless every input of one depends unconditionally on the output of the other.

The other half of the problem, keeping the number of frames reasonable, is addressed directly by k-bounding. In setting up a loop, k activation frames are allocated and linked together, as explained in Appendix A. By setting the k-bound no larger than the useful unfolding of the loop, we ensure that most of the allocated activation frames will be in use. Storing loop constants into activation frames is simply part of the set-up operation. The iteration identifier disappears from the tag and, instead, we have <code-base, instruction-offset, frame-base, operand-offset>. An acyclic code-block invocation uses a single frame, while invocation of a loop uses k, the loop bound.[2] This line of evolution has been carried forward elegantly by Papadopoulos in the Explicit Token-Store architecture[74] and is implicit in recent hybrid architectures[55, 72].

9.1.3 What's in a token

This shift from names to addresses brings with it a change in what flows through the graph: tokens carry not just values but resources. Portions of the graph perform computations related to resource management, rather than to computing the answer. This change started with the introduction of I-structures, since an array of slots is explicitly allocated and filled. The allocate operation produces a descriptor for the region of storage for the structure. With tags denoting resources, there is also an allocate operation for activation frames and a similar descriptor carried on tokens. With k-bounding, the graph contains code that computes how resources are to be allocated and recycles resources explicitly in the graph. The arcs introduced to control loop unfolding are essentially *anti-dependence* arcs: artificial dependencies introduced to allow for controlled reuse of resources.

This shift has an impact on the compilation paradigm also. At the very least, the presence of side-effects against resources complicates certain optimizations. On a deeper level, the value produced in allocating a context for a loop should be viewed as an *object* that can accept various commands. One command is to initialize itself for execution with k concurrent iterations. Another might be to establish loop constants in each frame, although these two operations may

[2] Although this new execution model can be formalized without introducing an explicit notion of processor, to realize it efficiently an activation frame must reside on a single processor or small set of processors.

be combined. Invoking the loop is a similar command, as is triggering the loop to deallocate itself. Each of these commands is acknowledged so that a sequence of them can be chained together.

We may also want to associate the resources used by an iteration directly with the frame, much like a loop constant, but with *different ones in each frame.* For example, the object corresponding to an inner loop invocation may be associated with a particular activation frame in the outer loop. Thus, the setup for the outer loop builds a ring of frames and invokes the set-up for the inner loop for each, building a ring of rings. In this case, we may want to support several loop constant commands, so that inner-loop variables that are also free in the outer loop need only be stored once and may be used by several inner loop invocations. Other resources, such as I-structures, may be associated with particular frames as well, or passed between iterations. Thus, tokens carry references to objects, in addition to values, and objects may support a variety of commands, with some protocol for sequencing commands.

With resources and resource management code both explicit in the graph, we have meaningful control over the space complexity of a program. For example, many scientific applications involve a recurrence over some collection of large data structures. The programmer may want to control how many versions of the data structure co-exist, and can do so by specifying the k-bound on the loop forming the recurrence. Control of activation frame usage requires more detailed constraints and is probably best stated as a high-level requirement, with the details resolved according to the policies presented here.

Where the structure of the program is very simple, we may even want to consider merging resources to produce very efficient code. For example, in a loop that performs a simple recurrence over a structure, we may want to allocate the structure *within* the activation frame for each iteration. If the access patterns are very simple and the bounds are known, we may want to unroll the loop and remove fetches by holding components of the structure in slots representing arcs. This is not within the current state-of-the-art in dataflow compilation, but is an interesting direction for future work.

9.2 Use of the Ideal Execution Model

The primary problem addressed in this thesis is one of determining how the k-bounded loop mechanism can be employed in real programs. By focusing on the behavior of programs under the ideal execution model, rather than under the complex set of influences present in any real machine, we were able to make headway on the development of k-bounding policies for dataflow programs. An important direction for future work is to verify that these policies are effective on real machines, and that the guidelines developed here are consistent with program behavior on such machines. As a first step, we reflect on the development here and examine where the results may break down.

9.2.1 Latency

First, let us consider the effects of communication latency. If every token were subject to the same fixed communication latency, l, we would expect to see little qualitative effect on program behavior other than a kind of time dilation. All the operations in a step under the ideal execution model would fire, then tokens would be in transit for l steps, and then the next step would take place. The exception to this is that *I-fetch* operations should be subject to twice the latency penalty, since two messages are sent. This introduces a small, but potentially pervasive, change in the execution schedule. This change is even more pronounced if we suppose that only I-structure operations experience communication latency.

The presence of communication latency increases the amount of parallelism required to keep a given number of processors busy. However, the approach here has been to expose as much parallelism as possible under a given resource constraint, so the policies should still be applicable. However, the details of some of the analysis are affected. The most apparent effect is in estimating the useful unfolding of a loop, *cf.* Chapter 4. In the presence of communication latency, it may take longer to traverse certain arcs than others. This can have a significant effect on the behavior of innermost loops. For example, if latency were experienced only in the *I-fetch* operations in our inner product example, the duration of each iteration, c, would increase but the initiation rate, i would be unchanged. Thus, the natural unfolding of the loop, $\mu = c/i$, would increase. By allowing a larger k, the loop is able to sustain a larger number of outstanding memory requests, so the increase in the critical path length is small, in spite of the

k_o									
10	21	[25]	[28]	[32]	[33]	[34]	[35]	[36]	36
9	14	17	20	24	25	26	27	28	28
8	14	17	20	23	24	25	26	28	28
7	13	16	20	23	24	25	26	27	27
6	13	16	19	23	24	25	25	27	27
5	[13]	[16]	[19]	[22]	[23]	[24]	25	[26]	26
4	10	12	14	[17]	18	18	19	20	20
3	7	9	11	[14]	14	15	15	16	16
2	[6]	[7]	[9]	[11]	11	12	12	13	13
	2	3	4	5	6	7	8	9	10

k_i

Figure 9.1: P_{ave} Over k-space for Relax_step', 10×10, with A_{max} Optimal Points Marked

increase in latency. The question is, At what point is the effect of latency significant enough to warrant a change in policy? In the inner product example, with zero latency we have $i = 4$ and $c = 9$, so the latency on *I-fetch* operations would have reach 10 before the natural unfolding of the loop would double. Of course, greater unfolding of the outer loop also permits a greater number of outstanding memory requests. Thus, it is unclear that the policy needs to change.

On the other hand, if latency were experienced on tokens that cross from one iteration to the next, as would be the case if the mapping to processors were on a per iteration basis, then both i and c would increase. For innermost loops where i is small, this will substantially reduce the useful unfolding. In the case where the loop has limited useful unfolding due to a strong recurrence, $c \approx i$, even if the computation in the body is large, the presence of latency will affect both components similarly and not change the unfolding of the loop.

As a simple experiment to gauge the importance of latency, we may consider the behavior of the simple relaxation program discussed in Section 5.3.1. Figure 9.1 shows the P_{ave} observed in executing this program under a modified execution model, in which a token sent to or from an I-structure cell experiences a latency of 5 time units. This means the result of an *I-fetch* is available 10 time units after the later of the fetch and corresponding store. We see that under this assumption the inner loop can unfold beyond the $\mu = 5$ estimated for the ideal case and demonstrated in Figure 5.19 above (Page 111). However, from $k_i = 5$ to $k_i = 10$ the increase in P_{ave} is still quite small, no more than 20% in this example.

The general shape of the parallelism and resource contours differs only slightly from the

ideal, zero-latency case. A small amount of inner loop unfolding is advantageous under tight resource constraints, but otherwise the outer loop should be favored, while avoiding a partial "second wave." Increasing the latency not only increases the potential unfolding of the inner loop, it increases the effectiveness of outer-loop unfolding by increasing the duration of each iteration.

Deferred reads effectively introduce long latency in the response to *I-fetch* requests, even under the ideal model. They may cause even a simple loop to unfold arbitrarily, in the absence of k-bounding. Deferred reads are plentiful in several of the examples studied here, and yet the policies still proved effective. In the relaxation examples, each step issues reads against the version of the mesh produced by the previous step and many of the reads may be deferred. All of the producer/consumer relationships studied in the context of adjacent loops give rise to deferred reads. An avenue of future work is to consider the effects of deferred reads more carefully. Through detailed analysis of the graph, we can estimate the benefit due to allowing greater unfolding of loops in the presence of deferred reads, with certain assumptions on the pattern of corresponding stores. For example, in the inner product loop shown in Figure 2.3, assume that all the *I-fetches* are deferred until some time and then instantly they all are satisfied. With the loop tightly bounded, the time to completion after the fetches are satisfied is essentially $4n$ steps. This is determined by the circulation of the index variable. With the loop allowed to unfold completely, this may be reduced to $2n$ steps. This is determined by circulation of the accumulation variable. For intermediate k-bounds, the time to completion is at least $4n - 2k$. Under more realistic arrival patterns, the benefit of large k-bounds may be even smaller. Such initial studies in this direction, as well the quality of results obtained by ignoring deferred reads in policy formulation, indicate that the return is likely to be small.

The presence of latency and deferred reads may suggests that inner loops with significant potential unfolding under long latency assumptions should be treated as UNfold-If-Necessary, rather than Limited Useful Unfolding. This increases the flexibility in the k-bound assignment for nested loops when resources are plentiful, but compromises the effectiveness of the threshold technique in handling adjacent loops.

9.2.2 Variations in execution time

A short-coming in our simple model of loop unfolding is the assumption that iterations are homogeneous, *i.e.*, that i and c are constant. In practice variations occur due to network and resource contention, as well as data dependence in the computations. Consider the following very simple example:

```
{for i from 1 to n do
  A[i] = f i};
```

The policies developed here would give the same amount of resources to each invocation of `f`. However, if the computation within `f` is highly sensitive to the input data, such a policy will starve some iterations while others are over-endowed. The k-bounding mechanism itself is not the problem, because fixing the number of concurrent iterations does not require that they all receive the same resources. The basic technique of augmenting the program with resource management code also does not require equal allocation to all iterations, although that is usually the simplest approach. The resource management code can get arbitrarily complex. The primary problem is determining that some sophisticated partitioning rule should be applied. Also, if we have such a rule, solving for k can be difficult.

However, if the computation within the loop can vary widely, it must also be fairly complex. It probably contains nested loops and even branching. The policies developed here respond somewhat to data-dependent behavior, as the bounds established for subordinate loops and branches can be data dependent. The Simple code does exhibit data-dependent behavior, as the duration of each table look-up and the number of table look-ups per point depends on data values at the point, unfortunately, the tables employed with the benchmark code are so small that the variations are relatively minor.

Variations in execution time are manifested under the ideal execution model, but in a more realistic model they can occur even when the computation is homogeneous. The k-bounded loop approach does not prohibit variable resource allocation to iterations, but also does not cater to it. It might be possible to establish a pool of resources to be used by the k concurrent iterations, where the distribution of resources over the iterations is dynamic. However, such a powerful mechanism comes at a significant cost and runs the risk of deadlock due to over-commitment of resources, much like the dynamic throttles, discussed in Chapter 1.

9.3 Compilation Techniques

In this thesis we have focused on development of policies for k-bounding; the analysis techniques were implemented as necessary to carry out a policy. This included automatic construction of call trees, propagation of R_{min} values up the tree, symbolic analysis of iteration counts, critical-path length analysis, and worst-case token storage analysis. The synthesis aspect, augmenting the program with code to determine k-bounds, was performed manually. However, an important sub-goal of the work is to understand what compiler analysis is necessary to implement reasonable policies and what additional gains are possible with more sophisticated analysis. The bottom line is that the better the analysis, the better the policy. Fairly simple analysis goes quite far, but will miss non-trivial dependencies. Where we fail to detect the presence of a dependence, we may err in the direction of investing resources where little use can be made of them. In this section we discuss various kinds of analysis, roughly in order of decreasing importance.

Determining the maximum useful unfolding of innermost loops is crucial. These loops are usually nested quite deeply, so any resources invested there are multiplied many fold. For the kind of inner-loop constructs that occur frequently in practice, straightforward graph analysis is sufficient. In the presence of conditionals and dependencies through data structures, more sophisticated analysis is required. Still, fairly crude analysis is adequate, since it can only affect performance, not correctness. For example, with conditionals we need to produce a reasonably good estimate of the path lengths through the conditional. It may be possible to weight the two arms with a probability estimate, as in Paraphrase[76], or make worst-case assumptions. Traditional subscript analysis can be applied to detect dependencies through structures[87], although for innermost loops it generally is not necessary.

Detecting outer level loops with limited useful unfolding or expensive unfolding is also crucial. With an Outer-In bias, failure to detect such a situation can result in a very poor policy, *i.e.*, most of the resources are invested where little use can be made of them. In scientific applications such loops usually involve structure recurrences. Although it may be difficult to determine the exact nature of the recurrence, it is straightforward to determine that a loop contains a structure allocation. In such cases, a version of the structure will exist for each concurrent iteration. Usually it is not difficult to derive a simple symbolic expression for the

size of the structure, although determining that the structure is "big" may have to be heuristic. Providing a hard guarantee on structure storage usage requires determining the lifetime of structures, in addition to determining their size.

Analysis of iteration counts is very important and serves a dual role. Extensive use of array comprehension or higher-order functions can give rise to many loops with a small, constant number of iterations. These need to be recognized, in order to avoid investing resources there. It is also valuable to be able to form simple symbolic expressions for the iteration counts of various loops. Wherever this is possible, we can avoid assigning a k-bound greater than the iteration count. This is critical to the policies dealing with nested loops, as well as for the threshold splitting policy. Also, we can avoid k-bounds that give rise to a sparse "second wave". It very important to lift iteration count information up as far as possible in the call tree, because doing so enables policies that operate on multiple loop levels at once, such as Inner-Out.

On the experimental side, more study is required to fully understand the implications of not being able to determine iteration counts, as in symbolic programs and sparse matrix routines, which employ list traversal extensively. Also, more study is required of programs with very dynamic behavior, such as numerical quadrature methods[58].

Sophisticated dependence analysis can improve policy determination in a variety of ways. The simplest form is inter-iteration dependencies via arrays which may further constrain the unfolding of an inner loop. More important is detecting strong recurrences in loops containing code-block invocations and, possibly, nested loops. The latter can be tricky in a non-strict language because the subordinate invocation may produce results before all the arguments are received and may continue executing after the result is produced. Thus, even the following trivial example it is non-trivial to determine the strength the recurrence:

```
{while p A do
  next A = f A
 finally A}
```

Clearly a new iteration cannot start before `A` is available, but the nature of the overlap depends on the internal structure of `f`. If `A` is a scalar produced by the final computation in `f` there is a very strong recurrence, but not if `A` is an I-structure returned from `f` while it is still being filled. In many cases where there is a strong recurrence, it is also easy to detect. For example, the final expression of a function may have a scalar result that is unconditionally dependent on

every node in the graph. However, a variety of trickier cases can arise, as when the outer loop reads an element of an array that is the the last value stored by an instance of the inner loop.

With fairly novel kinds of dependence analysis it would be possible to characterize structure recurrences better, and thereby arrive at better policies. For example, a class of structure recurrences are *local* in the sense that each point in a structure depends only on a local region of the previous version of the structure. Mesh relaxations often exhibit this kind of recurrence. The critical path of the computation producing a new point determines the useful unfolding of the loop with the recurrence. A different class of structure recurrences are *global*, in the sense that each point in a structure depends on essentially all points in the previous version. This can arise through convergence tests, step-size calculations, and the like. Detecting it may require tracing dependencies through several structures and phases of the computation.

The data presented here indicates that substantial gains are possible from traditional loop optimizations, such as static unrolling and loop interchange. By virtue of the recurrence on the index variable, small innermost loops offer only modest parallelism. This suggests that, where possible, the strongest recurrence in a loop nest should be moved to the innermost level. Alternatively, the size of the body can be increased by unrolling the loop. This exposes more parallelism, often with little increase in resource requirements, *and* reduces program execution overhead[8].

The severity of the parallelism/resource problem is strongly affected by the degree of non-strictness permitted. We have seen that the problem is very severe for *Id* programs, where all data structures and functions are non-strict. The problem appears to be less severe and easier to manage in more restricted contexts. Certainly, it would be beneficial to determine when a function cannot make significant progress without a particular argument, as this can be used to defer initiation of the function. Also, it would be valuable to determine when a pair of adjacent loops can be safely serialized.

In summary, fairly simple analysis techniques adequately detect the most common factors limiting useful program unfolding, but very sophisticated analysis is required to detect the more esoteric constraints. In general, by failing to detect a dependence we assume two computations will execute in parallel when, in fact, they will not. This is in direct opposition to the situation with parallel execution of programs expressed in sequential languages. Thus, even though a variety of analysis techniques can be borrowed from that work and the ultimate goals of the

two approaches are similar, we will never arrive at the same point. We will err in the direction of presuming parallelism that is not present, while they will err in the direction of missing potential parallelism.

9.4 General Programs

The importance of k-bounding is most apparent in scientific applications, since they are usually iterative in nature, and we have focused on this fairly restricted class of programs throughout this thesis. In this section we consider briefly how k-bounding might be extended to manage parallelism in general programs. We certainly cannot make hard guarantees on resource requirements, but may still be able to apply some of the techniques. Two directions hold potential. One is based on the dynamic execution tree techniques discussed in Chapter 1, where the scheduling unit is a fairly large portion of the program, on which k-bounding is applicable. The second is to employ k-bounding as if we were within our restricted class of programs, and accept that no resource guarantees are possible.

9.4.1 Scheduling resource bounded graphs

The dynamic execution tree techniques for controlling parallelism discussed in Chapter 1 generally involve making decisions on a per code-block invocation basis. No special attention is paid to the structure of the program, so these techniques are fully general. On the other hand, because they ignore program structure, they are not very robust. Improper decisions can lead to deadlock. Recall, in our case study using the Simple code the call tree included nearly 300 nodes. The dynamic execution tree is immense! If decisions are made on a per code-block invocation basis, a huge number of decisions must be made. Most of them are trivial, but a few are very significant. We saw that static analysis rapidly focused attention on the critical portion of the program. Thus, it would seem that by combining the two approaches it might be possible to generalize loop bounding, while making dynamic execution control more robust.

Suppose we take a portion of the program, rooted at some function f, where k-bounding is applicable. This portion is analyzed and augmented with a new resource parameter, r. The augmented function, f', can be used in any context and will operate within resource constraint, r. We call such a program a *resource bounded graph* (RBG). Note that even though k-bounding

can always be applied from the bottom up, by considering a nest of loops in the context of use, we are able to make a better decision and may arrive at a different policy than we would by looking at the nest in isolation.

We can view a general program as including a collection of RBGs with some recursive backbone and can consider employing a dynamic scheduling approach (branch under low parallelism, not under high), where the scheduling quanta are RBGs. Thus, the decision we need to make is not just which object to schedule when, but also how much of the available resources are provided to it. It will manage its own resources and scheduling internally. Instead of making a large number of trivial decisions, we make a few well-considered ones.

9.4.2 k-bounding without guarantees

The other approach to dealing with general programs is to employ the k-bounding approach, even where the conditions for applicability are not met, *e.g.*, where the call tree cannot be constructed. The idea of passing in a resource limit and partitioning resources at branches under program control can be applied to any program. Where an RBG is entered, the computation within the RBG is self-managing, so unfolding is determined by the resource limit at the root of the RBG. At some level within the execution tree, the resource limit for a given branch becomes so small that no more unfolding can occur. However, the branch may continue to extend arbitrarily deep, so we can make no absolute guarantees on resource usage. We may want to use only a fraction of the resource initially, in case some branch exceeds its limit.

In formulating a policy for controlling unfolding in the general case we have less information to work with than in the restricted case studied here. We can no longer determine an R_{min} value for a given branch. We will need to guess one, in order to split resources at a branch. The threshold technique can be applied, where branches are simple enough to offer and R_{max}. Outer-In policies also require taking a guess at the minimum requirements per iteration. However, for a given r and k, we can proceed with determining loop bounds for the internal nests.

9.5 Summary

Resource usage is a serious concern for any parallel machine, because more parallelism generally means more storage. We have demonstrated the seriousness of this concern in the context of

dataflow machines and introduced a mechanism for coping with it, k-bounded loops. The mechanism is attractive because it is effective and makes machines *easier* to build. There is always a danger in computer architecture that an elegant mechanism will prove impossible to employ in practice. From the work presented here, we can expect to produce adequate k-bound assignments automatically for a reasonably large class of programs. Simple guidelines have been laid down for dealing with nested loops of various forms and for partitioning resources among adjacent loops. Empirical studies suggest that policies based on these guidelines perform well, although final judgment will require examination of a broader, representative sample of applications on real dataflow machines of significant size.

Appendix A

k-Bounded Loop Machine Graphs

In this chapter, we pursue a more formal development of the k-bounded loop mechanism. We demonstrate certain conditions that guarantee proper bounded-loop behavior and develop a loop schema that meets these conditions in a simple manner. We also discuss trade-offs present in variations on this schema. The k-bounded loop schema is illustrated at a dataflow machine-graph level[85] first for the MIT Tagged-Token Dataflow Architecture[7] and then for an Explicit Token-Store dataflow architecture[74]. These machine graphs are stylized, but still rather primitive. Later, we characterize bounded loops at the more abstract program-graph level, which provides a framework for powerful optimizations. Finally, we demonstrate how structures can be recycled within a loop.

A.1 TTDA k-Bounded Loop Machine Graph

In this section we develop in detail the k-loop schema presented in Chapter 3. First, we establish general conditions for detecting completion of an iteration and using this to control initiation of later iterations. We discuss some of the trade-offs involved in various ways of meeting these conditions and describe a simple, efficient k-loop schema. Finally, we describe setup and cleanup for such loops.

A.1.1 Completion detection

If initiation of loop iterations is to be dependent on the completion of logically prior iterations, completion of each iteration must be detected. It is not enough that the values produced by an iteration are available, because some nodes may not produce an output, *e.g.*, the *switch* for a

value used in only one side of a conditional or an operation with side-effects may fire without producing any result tokens. Thus, we must embellish the graph to handle these cases.

Definition 1 *A loop iteration* is complete *when all operations for the iteration have fired and no tokens or deferred-reads remain in existence for the iteration. An iteration starts when the first token is produced for the iteration.*

To detect completion we add a *signal* output to every node that has no meaningful output and reduce the collection of signals in a block to a single overall signal. Details of this process are explained in [26] and [85]. For loops this means the predicate and the body may each produce a signal. If either does, an extra *L-switch* is provided and the signals are brought together as indicated in Figure A.1 by the portions of the graph shown in bold.

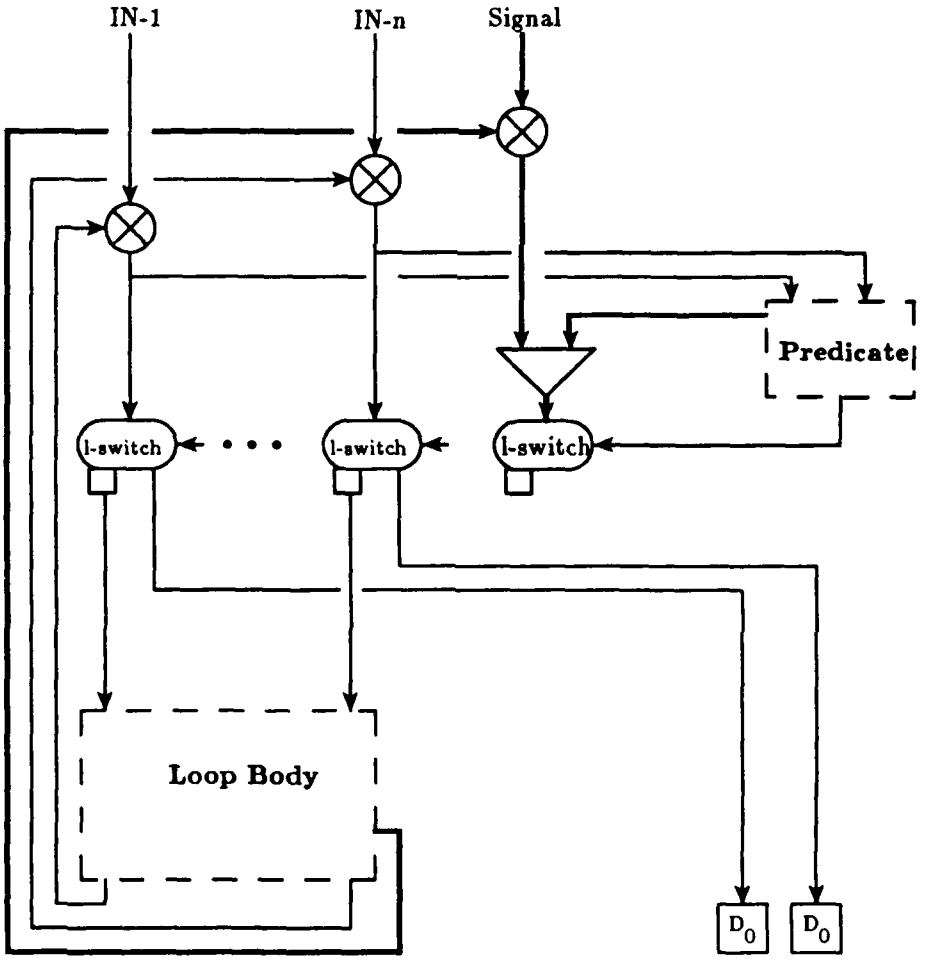


Figure A.1: Dataflow Loop with Signals shown in Bold

Theorem 1 *For a loop as in Figure A.1, suppose all the L-switches fire producing tokens with iteration number i, then iteration $i-1$ and all previous iterations are complete.*

By construction, every node in the loop is on a path from an L-switch to an L-switch. □

There are many ways to determine when all the *L-switches* of an iteration have fired, so we define a general condition for termination detection. First, we need to introduce some terminology. In the various examples shown in the thesis, nodes in the dataflow graphs have one or two sets of result arcs. More precisely, we say a node has one or two *outputs*, each of which may have several result arcs connected to it. When a result is produced for an output it is placed on every arc connected to the output. A set of outputs satisfies $Term(i)_j$ if producing a result at each output for iteration j implies that iteration i is complete.

Corollary 2 *Any set of outputs that covers the L-switches, i.e., for any program input the set of outputs are dependent on every L-switch, satisfies $Term(i-1)_i$.*

The TRUE outputs of the *L-switches* satisfy this condition trivially. They can be reduced to a single signal via a tree of *no-ops*, called a *completion tree*, as was indicated in Chapter 3 and as shown in in Figure A.2. However, with a little graph analysis it is usually possible to find a smaller collection of outputs, farther inside the loop body, that unconditionally cover the *L-switches*. This reduces the size of the completion tree, but, as we will see later, reduces the utilization of the resources allocated to the loop.

A.1.2 Initiation control

Given a means to detect when an iteration is complete, this event can be used to trigger the initiation of a later iteration. As a general condition for initiation control, a set of outputs satisfies $Gate(i)_j$ if no *L-switch* can fire producing a token for iteration i without a token having been produced on one of the outputs for iteration j.

There are various ways to locate a set of arcs that cover the *L-switch* inputs, however, by exploiting the structure of the loop schema itself this can be done trivially. Iteration $i+1$ cannot begin until the result of the loop predicate for iteration i is produced and delivered to the control inputs of the *L-switches*. By simply withholding delivery of the predicate value, initiation of further iterations is inhibited.

Corollary 3 *The output of the predicate satisfies* $Gate(i+1)_i$.

To constrain a loop to have bounded unfolding of some amount k, it is enough to tie $Term(i)_{i+1}$ to $Gate(i+k)_{i+k-1}$. This can be accomplished by inserting a D_{k-2} operator, which increments the iteration number by $k-2$, between a termination tree that detects firing of the *L-switches* and an *L-gate* that prevents firing of the switches, as shown in Figure A.2.

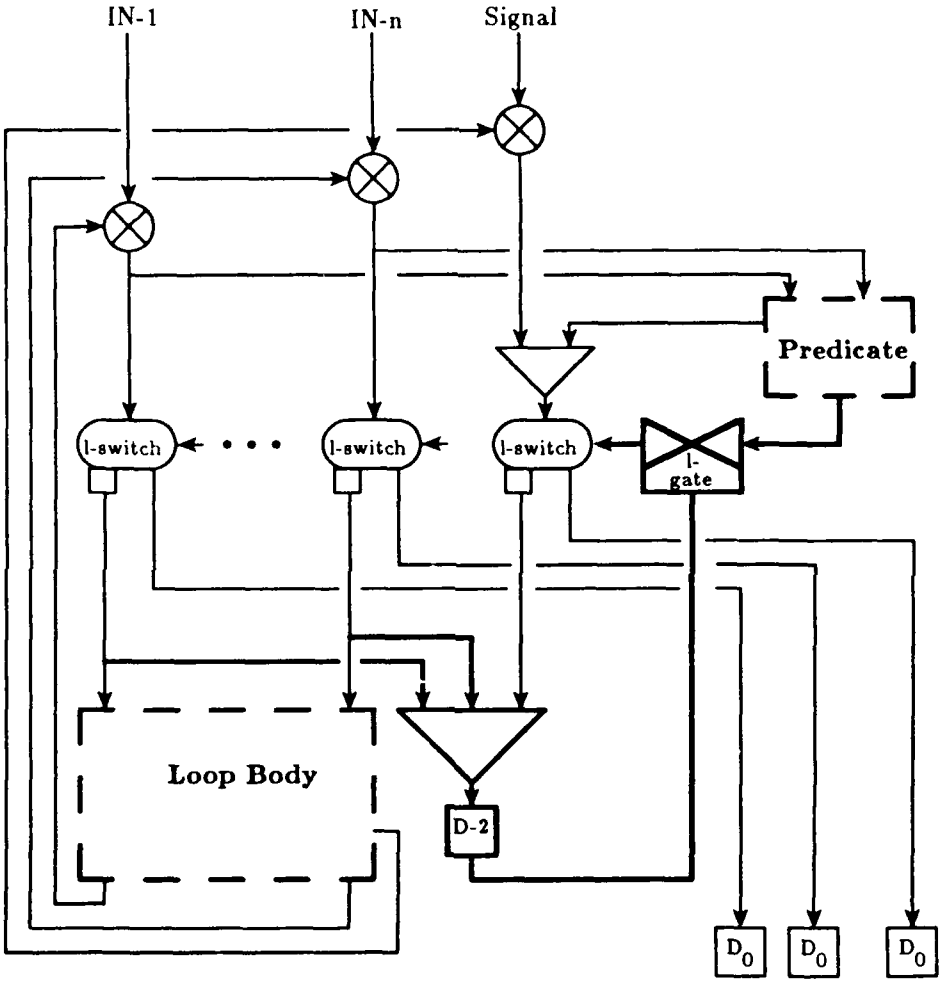


Figure A.2: Bounded Dataflow Loop

Definition 2 *Given a loop L, a* k-bounded loop *for L is formed by taking a set of outputs satisfying* $Term(i)_{i+1}$, *reducing them into a* D_{k-2} *operator, and using the output of that node to gate a set of outputs satisfying* $Gate(i)_{i-1}$.

A.1.3 Iteration reuse

Now that we have built up the basic mechanism, we need to show that loop unfolding is controlled in the manner we expect, *i.e.*, that a k-bounded loop can unfold to exhibit k concurrent iterations and no more. Also, it should be possible to execute such a loop with resources for k iterations, where these resources are reused in a natural way.

Theorem 4 *Given a k-bounded loop, initiated with triggers for iterations 0 through $k-2$ at the L-gate and initial input tokens for the loop variables, at most k consecutive iterations can be active at any time, where $k \geq 2$.*

> *We must show iteration $i+k$ is initiated only after iteration i completes. Iteration $i+k$ starts when an L-switch fires with inputs in iteration $i+k-1$. This requires that the D_{k-2} fired with input in iteration $i+1$, which in turn requires that every L-switch fired with inputs in iteration i.*
>
> *Since the loop predicate in iteration i must be evaluated before $i+1$ may begin, loop iterations are initiated in logical sequence.* □

Since only $k-1$ trigger tokens are provided, it might seem that there can be only $k-1$ iterations active simultaneously. In fact, there is a region of the graph where k iterations can be active simultaneously and a region where only $k-1$ iterations can. Let *recycle* be the cyclic graph formed by the portion of the graph reducing the *L-switch* outputs to the D_{k-2} and the portion connecting the D_{k-2} to the inputs of the *L-switch* inputs. At any point in the execution of the loop there are exactly $k-1$ waves of tokens in *recycle*. Initially $k-1$ tokens are provided as trigger inputs to the *L-gate*. Each of these allows a wave of tokens to flow through the *L-switches* and through the termination region, eventually producing a new trigger input with iteration number $k-1$ larger.

Nonetheless, k iteration numbers may be in use simultaneously. Let us consider a simple example. Figure A.3 shows the graph for the following simple loop.

```
FOR i <- 1 TO n DO next sum = sum + (f i);
```

This has two *L-switches* corresponding to `i` and `sum`. Suppose that the *L-switch* for `sum` does not fire. Initially $k-1$ trigger tokens are placed on the iteration control arc, and the predicate depends only on `i`, so $k-1$ waves may pass through the *L-switch* for `i`. The tokens on the

outputs of the firing *L-switch* have iteration numbers one larger than the tokens for the same wave waiting at the inputs to the other *L-switch*; thus the $k-1$ waves use k iteration numbers. This shows that k iterations are active, but not in an interesting sense. The following result shows that all k iterations can be doing useful, parallel work.

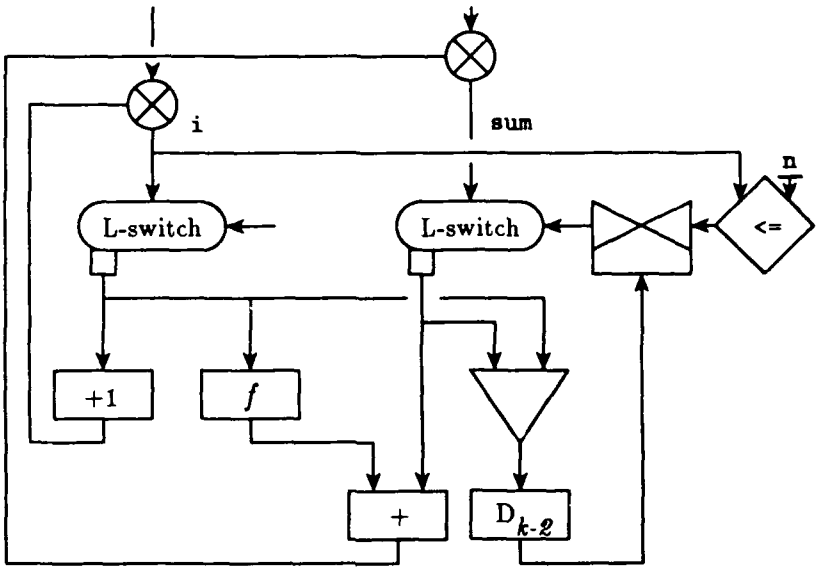


Figure A.3: Graph for a Simple Summation Loop

Theorem 5 *A k-bounded loop can have as many as k instances of a node enabled simultaneously.*

Consider the loop discussed above and pictured in Figure A.3, where n is assumed to be larger than k. Assume the node corresponding to `f` *does not fire. The first* $k-1$ *iterations can be initiated by virtue of the initial trigger tokens, so loop variable* `i` *circulates* $k-1$ *times, enabling* $k-1$ *instances of* `f` *with iteration numbers 1 through* $k-1$*. Once the initial value of* `sum` *passes through the L-switch, completion of iteration number* 0 *is detected in iteration 1 and a trigger with iteration number* $k-1$ *is produced. This allows* `i` *to circulate again and* `f` *to be enabled for iteration* k*.* □

As a more realistic scenario, suppose `f` is a function or expression that takes a long time to compute, compared to the rate at which the index variable is incremented. The same reasoning shows that under fair scheduling k waves of tokens will be passing through `f` concurrently.

Observe that if termination were detected using the output of the + no termination tree would be required. However, it would not be possible to have k instances of f enabled concurrently, since f would be part of the recycle region, in which only $k-1$ waves can be active.

The above results imply that in a k-bounded loop all arithmetic on iteration numbers can be performed modulo k. In a Tagged-Token Dataflow architecture, this means the iteration field on the token can be kept to a fixed size. Iteration numbers are effectively recycled within the graph. Moreover the utilization of resources is good, in that all k resources can be in use simultaneously.

A.1.4 Loop setup

To start a k-bounded loop, trigger tokens carrying iteration $0, \ldots, k-2$ must be deposited on the iteration control arc. This can be accomplished by the simple loop shown in Figure A.4. The value of k is provided as input with iteration number 0. The *set-k* operation associates k with the invocation, so that iteration numbers can be manipulated modulo k. It is important that completion of this setup graph be implicit in the production of all the triggers, and to meet this condition in a simple manner, we have employed a new operation, *G-switch*, which is a variant of *L-switch*. It receives a data input and a boolean control input. If the control input is TRUE, the value on the data input is produced on the left output, with the iteration number incremented. Also, the data input is produced on the right output, without change to the iteration number, regardless of the value of the control input. Thus, the *G-switch* produces two outputs with a TRUE control input and only one with a FALSE input.

Observe, the first iteration of the loop can begin as soon as the first trigger is produced. Further triggers are produced in the order that further iterations require them, thus the production of triggers overlaps nicely with the early iterations of the loop.

A.1.5 Loop cleanup

The loop graph we have constructed thus far correctly executes as a k-bounded loop, but is not self-cleaning; $k-2$ trigger tokens are left on the iteration control arc after the loop finishes. Removing these tokens is a bit involved and requires that the behavior of the *L-gate* be more complex than what we have described. Suppose the iteration number on the tokens emitted from the FALSE outputs of the *L-switches* carry iteration number i. We need to ascertain exactly

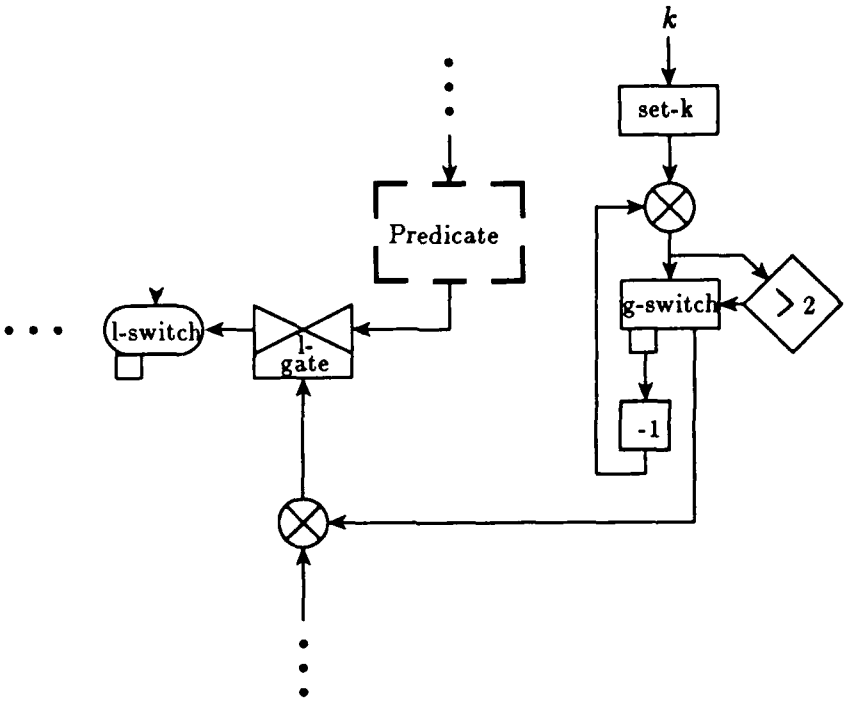


Figure A.4: TTDA k-bounded Loop Setup

what tokens remain in the graph. If the *L-gate* had not fired for the final predicate value, the iteration control arc would eventually contain $k-1$ triggers starting with iteration number i. In the final firing of the *L-gate*, the trigger token with iteration number i is removed, so eventually the iteration control arc will contain tokens carrying $(i+1) \bmod k, \ldots, (i+k-2) \bmod k$.

To remove tokens from the iteration control arc, we allow the *L-gate* to have an alternate output, as indicated in Figure A.5. It becomes a variant of the *switch*. If the input value is either TRUE or FALSE, the value is produced on the normal output. Otherwise, it is produced on the alternate output. Thus, the cleanup loop shown in Figure A.5 can remove the trigger tokens by sending integer values to the *L-gate*. The cleanup loop is initiated by a token from the FALSE output of one of the *L-switches*, providing the final iteration number. The cleanup completes only after all the trigger tokens have been removed by matching with tokens generated by the cleanup loop. To ensure that the k-loop schema is well-behaved, completion of the cleanup loop is gated into the normal loop signal.

Observe that the cleanup loop starts removing tokens as soon the *L-switch* fires for the loop exit. At this point, the final k iterations may still be in various stages of completion. The cleanup loop consumes triggers in sequence (oldest first), so if iterations terminate in the order

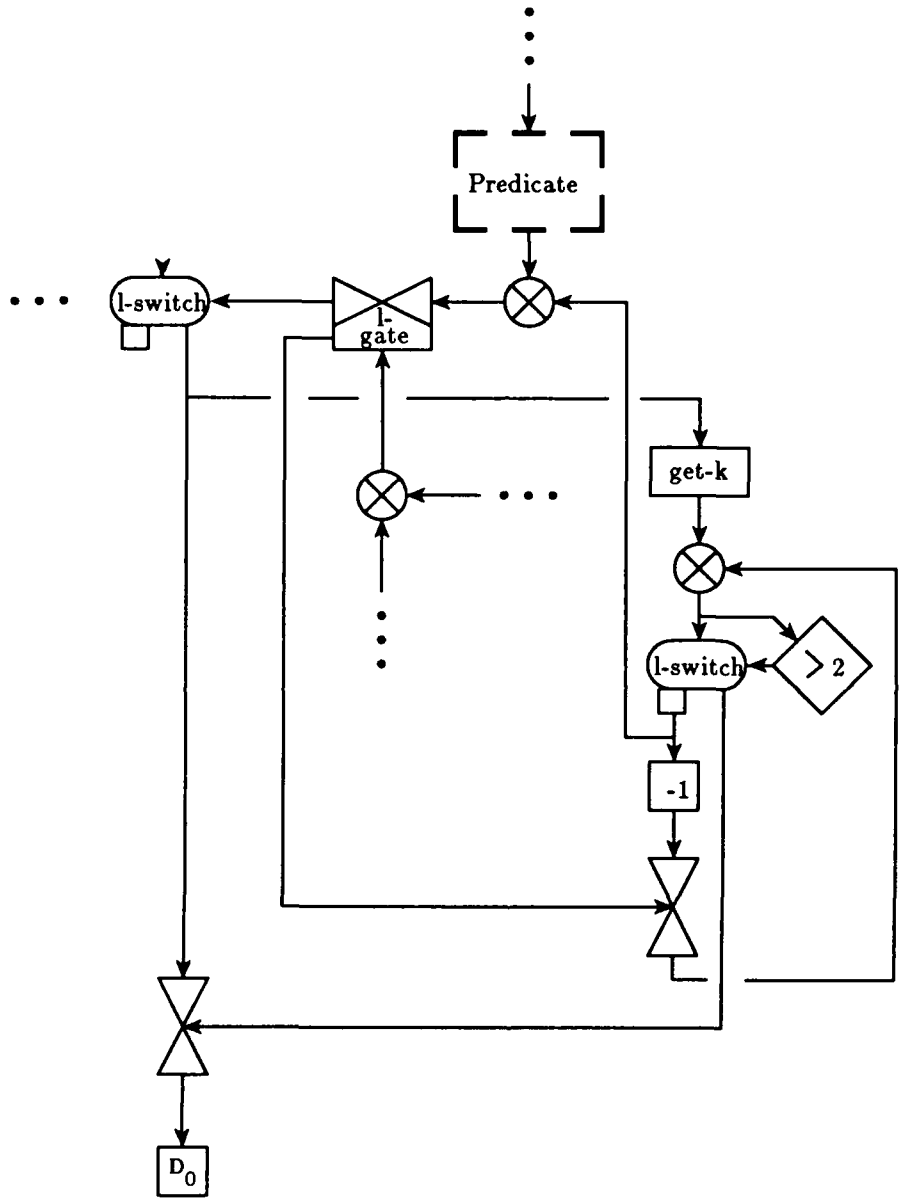


Figure A.5: TTDA k-bounded Loop Cleanup

that they were initiated, the removal of trigger tokens overlaps nicely with the final iterations of the loop. This loop schema is used in compiling all the examples presented in the thesis.

A.2 Monsoon (ETS) k-Bounded Loop Machine Graph

In an Explicit Token Store (ETS) dataflow machine[74], a k-bounded loop is realized by allocating k activation frames and forming them into a ring. Arithmetic on iteration numbers corresponds to sending tokens to the frame a certain distance around the ring. In this section, we develop the detailed k-bounded loop schema for the Monsoon architecture, analogous to that developed above for the TTDA.

The tag carried on a token in this architecture identifies the instruction that is to be executed, IP, and the location that is to be accessed in performing a match, fetch, or store, FP. The basic technique for detecting termination and controlling initiation in the ETS framework is essentially like that for the TTDA. The primary difference in the ETS graph is the explicit interaction with the activation frame. Rather than generate $k-1$ trigger tokens, we must allocate k frames. Each frame contains four special values: the tag for the return point (**Parent**), a frame pointer for the logically previous iteration (**Prev**), a pointer to the current frame (**Self**), and a frame pointer to the logically next iteration (**Next**). In effect, these form a doubly-linked ring, with each frame holding a pointer to itself and to the caller of the loop.

The ETS k-loop graph is shown in Figure A.6. The graph looks like a "repeat" loop at first glance, because the body appears above the predicate. Actually it is not; the predicate is evaluated on the initial inputs, before values enter the loop body. The orientation of the figure is shifted from that in the TTDA figures to reflect loop behavior on an ETS machine better. The bank of *L-switches* in the TTDA loop forms the boundary between iterations; the iteration field is incremented on the tokens sent into the body. Accordingly, *L-switches* are replaced by *send* operations in the ETS framework, which send the value appearing on the left input to the tag, <FP,IP>, appearing on the right input. A small integer constant is added to IP so that the proper instruction in the receiving graph can be selected. The IP of the receiving instruction is indicated by dashed arcs in Figure A.6. As long as the predicate evaluates to TRUE, the tag input for the *send* operations will carry the frame pointer for the next iteration. When it evaluates to FALSE, the *sends* receive a tag carrying a frame pointer for the caller, causing

values to be returned from the loop.

The first three special frame pointers, **Parent**, **Prev**, and **Self** are treated as loop constants, established when the ring is built. They can be accessed by a *fetch* operation, which reads a specified slot in the current frame. The **Next** slot serves a role analogous to the trigger tokens on the iteration control arc in the TTDA graph. It is accessed by a *fetch-extract* operation which causes the read to be deferred until the slot is full, if the slot is empty, and empties the slot upon reading it. Observe, that while the **Next** slot is empty, no tokens are sent into the activation frame for the next iteration. If the slot is full, it contains a tag for the next frame and signifies that that activation frame is ready to be reactivated. In reactivating the next frame, the slot is emptied, thereby preventing additional activations until the current one is complete.

The *send* operation produces one token that is sent to the remote frame and a second token that is delivered to an operation in the current frame, signaling completion. As in the TTDA loop, these signals are reduced through a termination tree. Upon termination of an iteration, the previous frame must be informed that reactivation is permitted. Thus, the final operation in an iteration sends the current frame pointer, **Self**, to the **Next** slot in the previous frame, **Prev**. This is shown as a *Remote-Store*, which stands for a pair of operations: a *send* in the current frame directed to a *store* operation in the remote frame. This final operation is equivalent to the D_{k-2} in the TTDA graphs.

In effect, each frame has a special *handle*. Possession of the handle implies permission to activate computation within the frame by sending tokens to it. While the loop is executing, an iteration activates the next frame by sending values of the loop variables to it. After the ring is created, the caller should possess the handle for one frame in the ring, so it can activate it with the initial values of the loop variables. We will see that other kinds of activation are possible: initialize, update, and deallocate. The computation resulting from an activation detects its own completion and returns its handle in an orderly fashion, as we will see below.

When the predicate turns FALSE, the final values of the loop variables are sent back to the caller. In addition, a handle is passed to the caller, so that it can cleanup the ring of frames, or even reuse it. In either case, reactivation starts not with the frame in which the predicate turned FALSE, but in the next frame. Thus, once the handle for **Next** becomes available, it is returned to the caller as well, as shown in Figure A.6

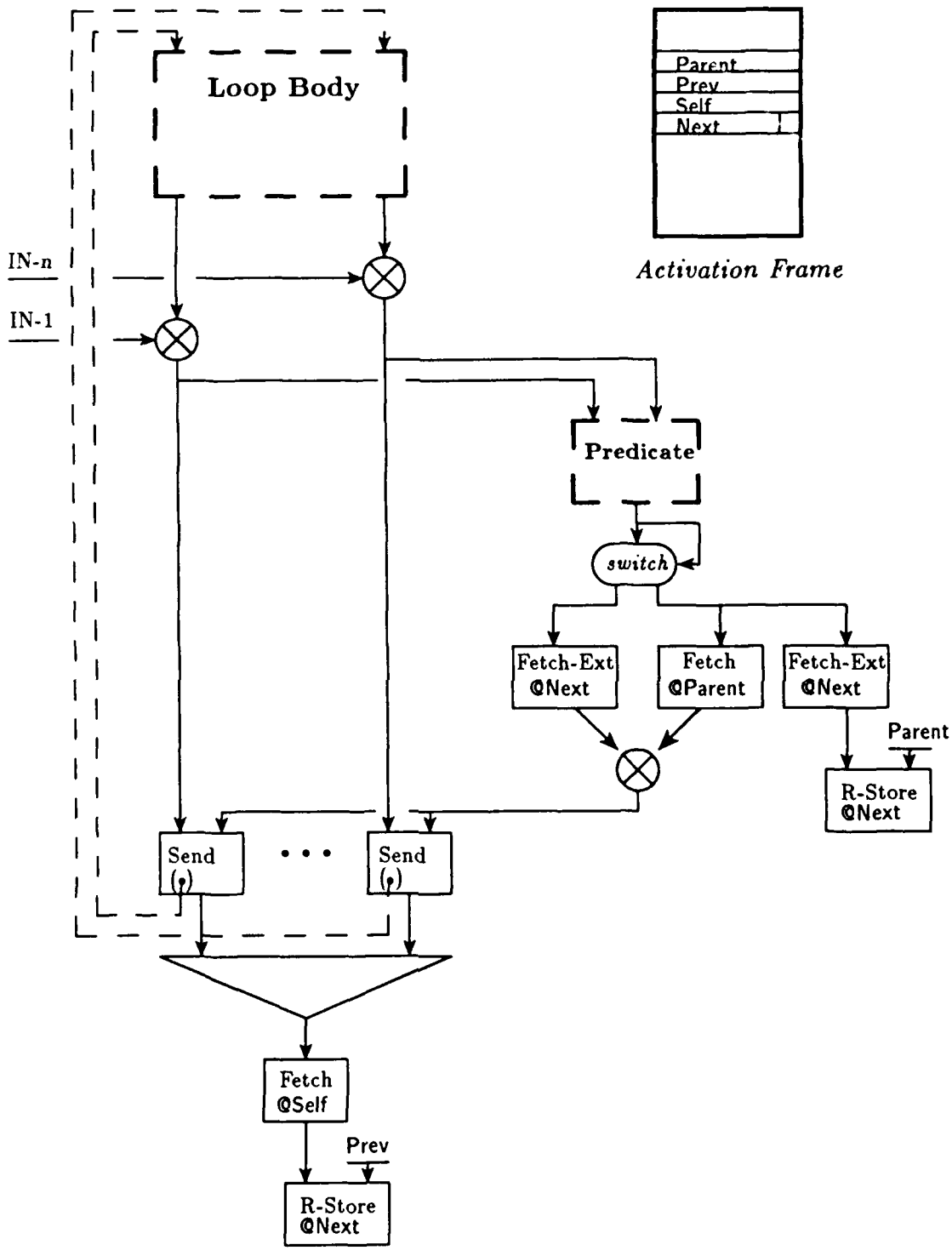


Figure A.6: ETS k-bounded Loop Schema

A.2.1 Loop setup

The setup process for k-bounded loop in the ETS framework is a bit more complicated than that for the TTDA, since frames must be allocated and links established. Also, obtaining a nice overlap is difficult, because the first iteration cannot determine its predecessor until the entire ring is constructed. (In the TTDA loop this is accomplished by simple arithmetic on the iteration number.) The idea behind the setup process is depicted in Figure A.7. Recall each frame for the loop is to contain three special constants (**Parent**, **Prev**, and **Self**) and has a reserved slot for the handle for the next frame. The caller of the loop allocates the first frame and provides the initial values for **Parent** and **Self**. For each of the k frames, all three of these special values must be stored before the handle for the frame is returned, indicating that the frame is ready to be activated. The setup for all but the last frame allocates the frame for the next iteration and sends the three special values to it. The last one sends the correct value of **Prev** to the first frame. Each frame detects completion of the setup operation, including storing the three special values, and returns its handle. For all but the first frame, the handle is sent to the **Next** slot of the previous frame; for the first it is returned to the caller.

The graph for the setup process is shown in Figure A.8. It is essentially a loop with six loop variables: the frame pointer for the previous frame (**Prev**), the return address for the frame handle (**RA**), the tag for the return point (**Parent**), the current frame pointer (**Self**), the frame pointer for the first frame in the ring (**First**), and the value of k. All but the first of these are supplied by the caller to the first frame. Until k is decremented to 1, a new frame is allocated and the appropriate values of the six loop variables are sent to it. In the final frame, the correct value of **Prev** is sent to the first frame. Each iteration detects its own completion and sends its frame pointer to **RA**.

While some overlap is achieved in setting up the ring of frames, the first iteration of the loop cannot begin until the entire ring has been allocated and the value of **Prev** has been supplied to the first frame. This is not necessarily a problem, because loops are often invoked long before their input data are available, but in certain cases it will add to the critical path considerably. To obtain better overlap, the value of **Prev** would have to be circulated during loop execution, not setup, so that the handle is returned when an activation is complete and the value of **Prev** is available. Thus, we can trade reduced setup time for an increase in the number of operations per iteration.

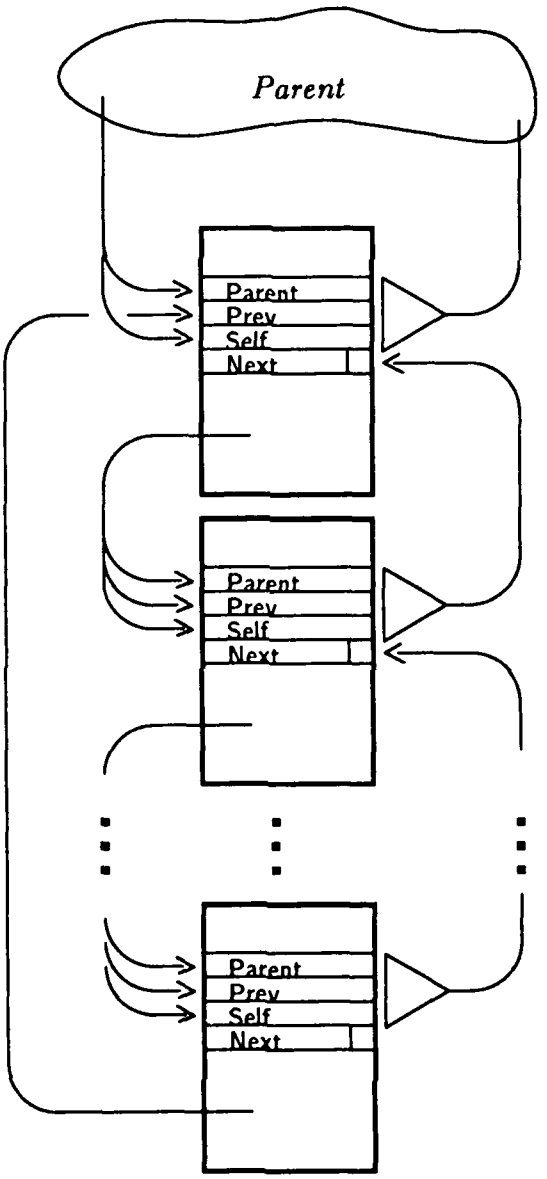


Figure A.7: Monsoon k-loop Setup Process

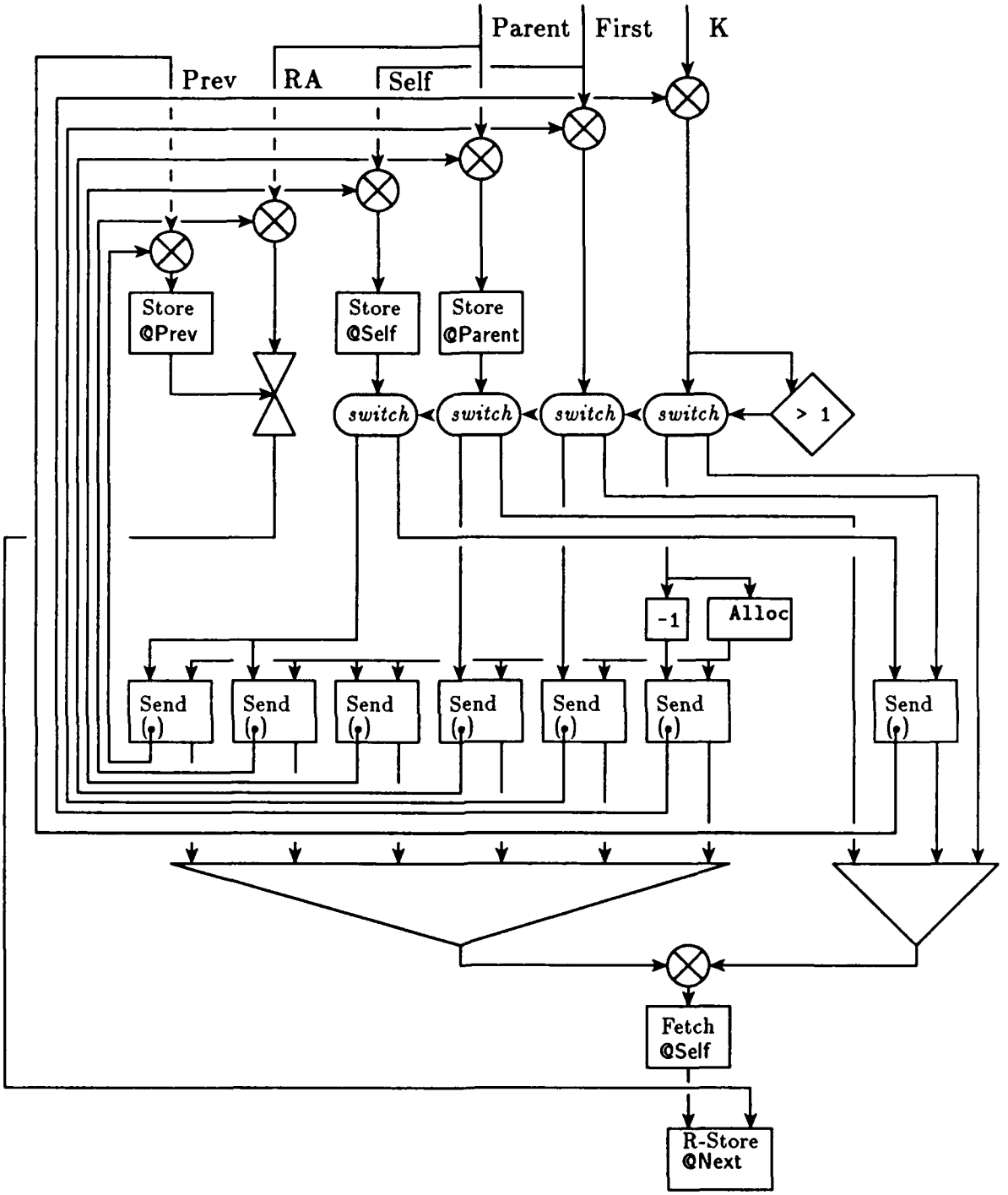


Figure A.8: Monsoon k-loop Setup

Loop constants can be stored into each activation frame in the ring for a k-bounded loop as part of the setup process. The setup graph is embellished with another input for each loop constant. The value is stored into the current frame, passed through a *switch*, and forwarded to the next frame, like the **Parent** value in the graph shown in Figure A.8.

A.2.2 Loop cleanup

Loop cleanup is fairly straightforward in the ETS framework, as shown in Figure A.9. The cleanup process traverses the ring of frames. Once the handle for the next frame is present, the cleanup signal is forwarded, and, finally, the frame deallocates itself. Since the cleanup process begins at the first frame after the one in which the predicate turned FALSE, cleanup overlaps nicely with the final iterations of the loop.

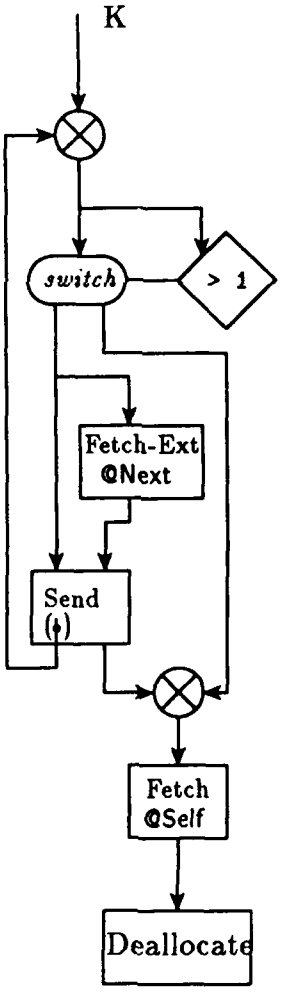


Figure A.9: Monsoon k-loop Cleanup

Like the TTDA k-loop schema, the ETS k-loop schema comprises the three graph fragments described in this section. Whereas in TTDA graphs these are tied together by common arcs, in the ETS framework they are tied together by common frame slots, namely, the four special values.

A.3 1-Bounded Loops

Our development of k-bounded loops thus far relies on the assumption that $k \geq 2$. However, there are a variety of circumstances where we may want *no unfolding.* The outer loop of Simple is one example — each iteration involves a huge amount of computation and tremendous amounts of resources. A little overhead to constrain the loop to $k = 1$ would be highly advantageous for a small machine. Some of the inner loops in this program demonstrate another important situation; they are small and have little useful unfolding, but are so deeply nested that a small amount of overhead is amplified dramatically. We observed above that there could be only $k - 1$ waves of activity in the recycle region, and in the schema in Figure A.2 this region is kept to a minimum. Pushing $Term(i)_{i+1}$ down toward the bottom of the loop reduces the effective unfolding of the loop to $k - 1$ iterations. This would improve the token storage, I-structure storage, and deferred-read behavior of Simple, although two iteration numbers (or activation frames) would still have to be allocated to the outer loop.

To execute a loop with a single iteration number, it is necessary either to introduce a second completion point and a second gate, or to form an equivalent loop with a single circulating variable. To explore the first option, suppose gates are placed on the arcs at the bottom of the loop. These gates are controlled by termination detection on the *switches.* (*L-switches* are not required, since $k = 1$.) This ensures that no token can enter the predicate until all the *l-switches* have fired, *i.e.*, until no tokens remain in the predicate. In addition, we build termination detection on the outputs of the gates at the bottom of the loop. This is used to gate the predicate. Thus, no token can enter the body until all tokens from the previous iteration have left it. The schema is shown in Figure A.10. For the outer loop of Simple this overhead is quite tolerable, since the amount of computation per iteration is so large. This schema has been implemented in the version of the *Id* compiler used for the single processor prototype of the Monsoon dataflow machine.

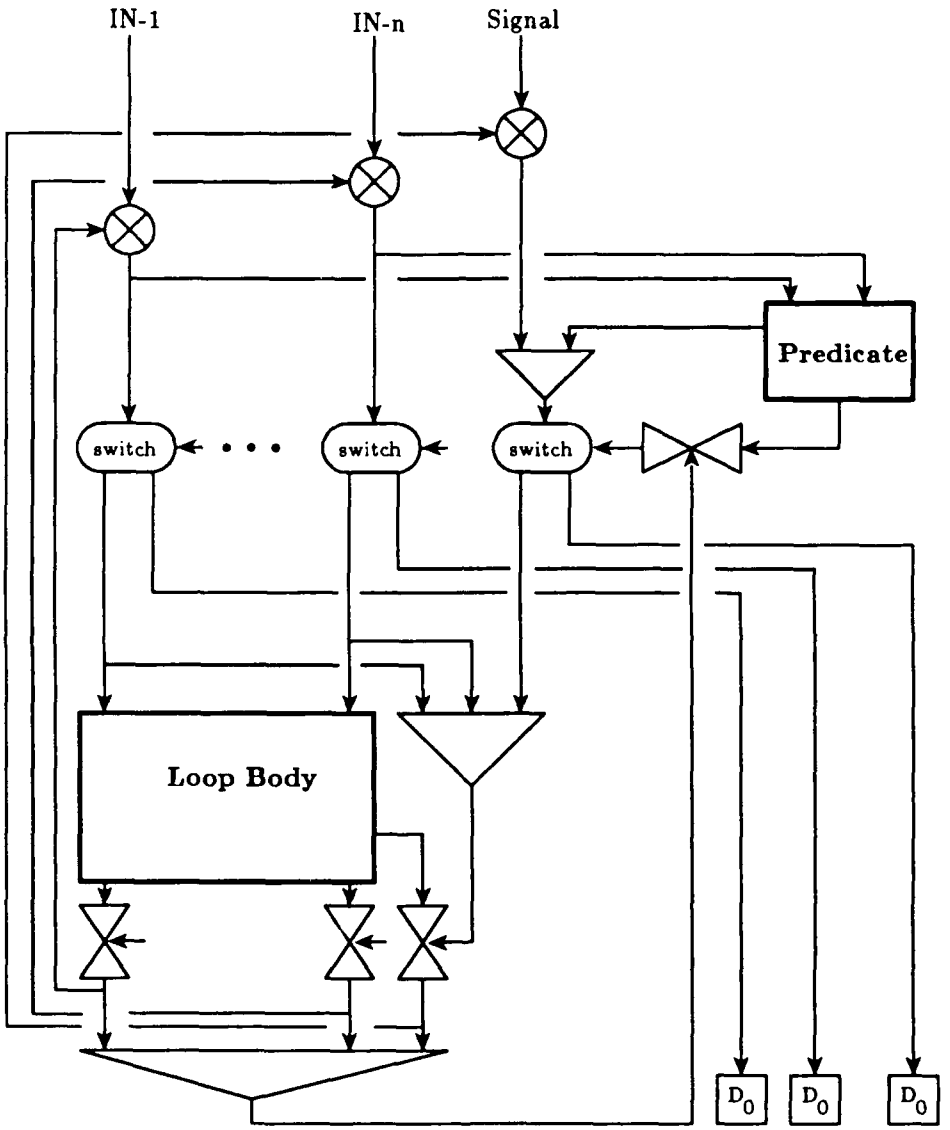


Figure A.10: 1-Bounded Dataflow Loop Schema

To explore the second option, notice that a loop with a single *l-switch*, including termination signals, satisfies the conditions discussed above for the 1-bounded loop, *a priori*. At the time a token enters the body, there can be no tokens in the predicate, since the *l-switch* fired. At the time a token leaves the body and enters the predicate, there can be no more tokens in the body. For a loop with a single loop variable and nodes that generate signals, we simply gate the signals generated in the predicate into the input of the *l-switch* and gate signals generated in the body into the circulating value. For a loop with multiple loop variables, we have to pack all the loop variables into a structure and tie signals into the remaining composite loop variable. In an ETS machine, we can store each of the loop variables at the bottom of the loop, and circulate only the signal that reflects the completion of the stores. For some of the inner loops in Simple that are nested deeply and are quite small, we will want to compile the tightest code possible and use whatever tricks the architecture allows.

A.4 k-Loops in the Program Graph

In developing the *Id* compiler, Traub introduced a powerful high-level representation of dataflow programs, called the *Program Graph*[85]. The various graph schema are represented by *encapsulators*, which preserve the hierarchical structure of the program, have well-defined semantics, and can be expanded into machine graphs for a particular dataflow architecture. Optimizations on the program graph are very powerful, because entire program constructs can be manipulated. This is particularly important when code motion is involved, as in hoisting invariants out of loops. The loop node in Traub's program graph does not take a position on whether loops are k-bounded or not. In generating a TTDA machine graph for a loop node, the k-loop schema described in Section A.1 is realized. This is perhaps adequate when the target machine is the TTDA because allocation of new iterations simply involves producing a trigger token carrying the appropriate iteration number. However, when viewing iterations as resources, as in an ETS context where each iteration has a corresponding physical activation frame, we would like to be able to represent and manipulate the allocation and deallocation of iterations at the program graph level itself. To this end, we outline a strategy for extending Traub's program graphs to provide a framework for expressing the resource allocation/deallocation aspect of k-bounded loops.

The loop node in the program graph has a pair of interior surfaces encapsulating the loop predicate and loop body, respectively, as shown in Figure A.11. The inputs to the predicate are the *nextified* variables and the loop constants. The nextified variables pass through to the middle arm, along with the result of the predicate. The inputs to the body are the nextified variables and loop constants. The outputs of the body are the nextified variables for the next iteration. The middle arm of the loop node essentially is translated into *l-switches* and the outer portion into the interconnection via *merges*.

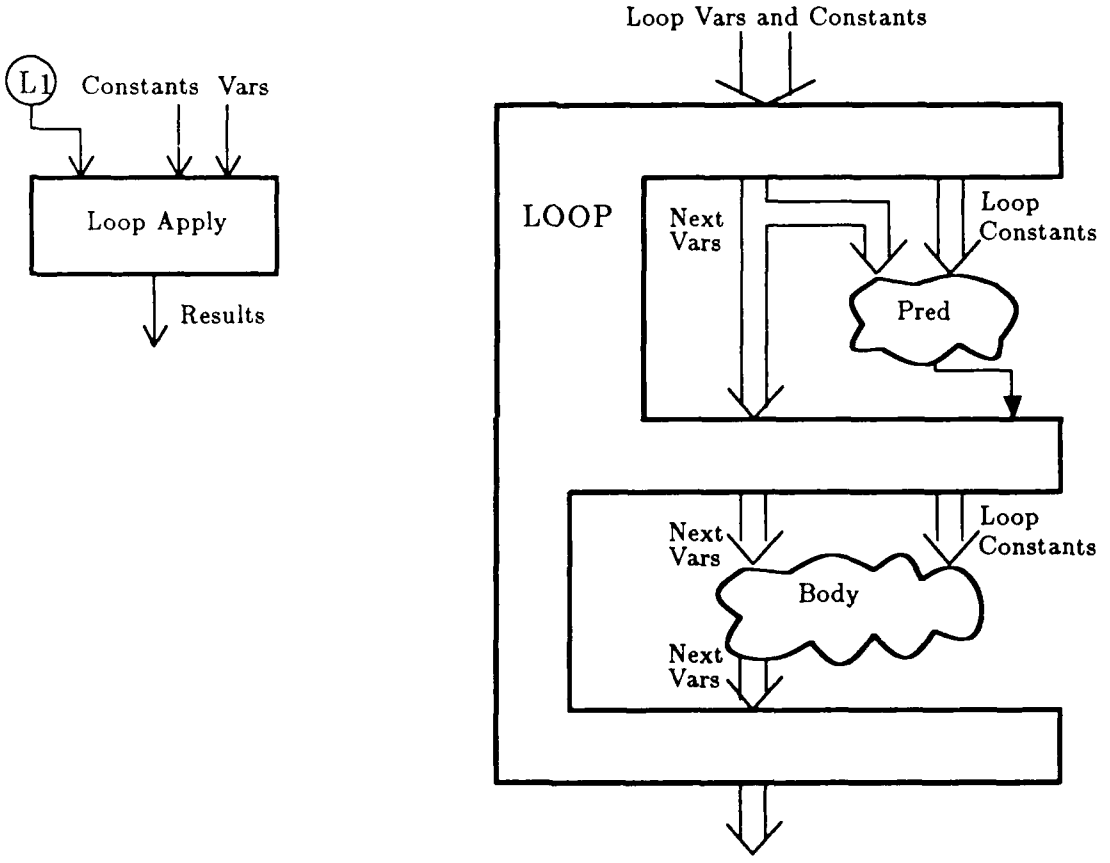


Figure A.11: Traub's Program Graph Loop Node

For a loop that executes n iterations, the predicate portion is activated $n + 1$ times and the body is activated n times. For k-loops there is also a portion of the graph that is activated k times prior to loop execution (setup) and a portion that is activated k times after loop execution (cleanup). To reflect this, the loop node is expanded to provide two additional interior surfaces.

As for the loop node, the minutia of interconnecting these graphs for a particular architecture is abstracted into the program graph node.

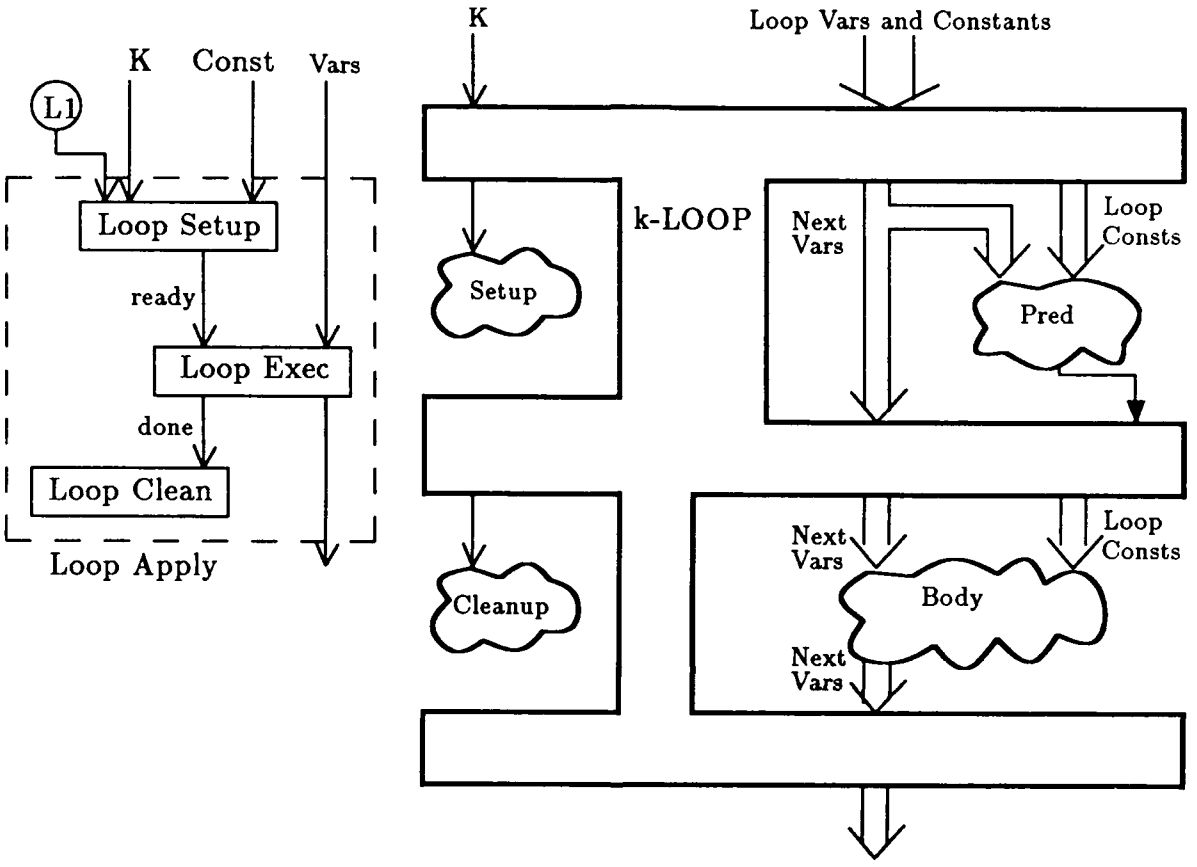


Figure A.12: Proposed Program Graph k-Loop Node

Initially, the setup and cleanup regions are empty, but as we perform optimizations and transform the k-loop node into a machine graph, code is moved into these regions. As a simple example, consider the step in which loop constants are transformed into activation frame entries. In the body and predicate, where a loop constant is provided as input to a node, the constant is replaced by a local *fetch* operation. The loop constant input is moved to the setup surface and connected to a local *store*. Thus, all the loop constants are stored in the activation frame during loop setup. If the slots for loop constants need to be reset before deallocating the frame, a corresponding *reset-slot* operation would appear within the cleanup surface. When the k-loop node is expanded into an application schema, the presence of the setup and cleanup surfaces

play an important role. Invocation of a loop is now a three-step process: setup, exec, cleanup. Thus, a loop-apply can be represented as a sequence of three operation; the output of the *loop-setup* is an initialized ring of frames and the output of the *loop-exec* is a ring ready to deallocated or reactivated. The loop constants appearing at the setup surface become inputs to the *loop-setup* node.

Once this separation is established, it is possible to lift the loop setup for a nested k-bounded loop into the enclosing loop, if the k-bounds are established upon invocation of the outer loop. The *loop-setup* node that builds a ring for the inner loop is shifted into the setup region of the outer loop. Thus, a ring of rings is formed. However, in doing this the loop constants for the inner loop may need to be stratified into two groups: those constant over the outer loop and those constant only over an invocation of the inner loop. Those in the first category can be stored during the outer loop setup. A vestige of the original *loop-setup* remains to store the second group of constants.

A.5 Structure Recycling

In Chapter 3, we introduced the idea of recycling data structures within the graph in a regular fashion. In this section, the details of this process are presented. We discuss a schema, the s-loop, for recycling structures built on top of the k-loop schema. It allows the number of structures to be controlled independently of the loop bound, which is important for certain kinds of experimentation, although usually the number of structures will be closely related to k. The basic strategy for structure recycling can be folded into the k-loop schema more intimately, but this requires that the setup and cleanup code be embellished significantly. The mechanism presented here was used to generate the profiles in Figure 3.6.

A central concept in the s-loop is the *structure bank*, an array holding up to s empty structures. (This functions essentially like the iteration control arc in the k-loop schema, but is initialized with a fixed number of empty structures.) A structure is withdrawn from the bank, filled, read, cleared, and finally deposited back. The loop forming the structure recurrence circulates a pointer, `rptr`, into the bank to ensure that structures are used in an orderly (deadlock free) fashion. The general form of the first-order s-loop body is shown in Figure A.13. The code for k-bounding is not shown, as it is independent of structure recycling. The schema

admits the following invariants:

1. The bank contains only empty structures.

2. If iteration i withdraws and fills structure j, structure j will be withdrawn before iteration $i+1$ withdraws structure $(j+1) \bmod s$.

3. At iteration boundaries, **rptr** points to a hole in the bank corresponding to the current structure and all other bank slots are determined to be filled eventually with empty structures.

A corollary of point 2 is that iteration i will withdraw its structure before iteration $i+s$ attempts to withdraw it.

The bank is initially filled with empty structures, except for a hole in the position referenced by the initial value of **rptr**. The initial structure will fill this hole after its first use within the loop is complete. The loop circulates some collection of loop variables, including a structure, **X**. One additional loop variable is introduced, **rptr**, which specifies the hole in the bank corresponding to the current value of **X**. The bank is assumed to be in the loop constant area. **rptr** is incremented, modulo s, to get the position of the structure that is to be filled, **wptr**. **wptr** is used to withdraw an empty structure from the bank. This is fed into the heart of the loop body, which transforms a structure into a new structure of the same size and shape, where an allocate would be in the standard formulation. The newly withdrawn structure becomes **NEXT X**. The graph for the transform produces a signal **dead X**, indicating that all reads and writes against the old structure **X** have been issued. This is used to trigger the clearing of that structure. When the structure is cleared, it is deposited at **rptr**. (A given structure cycles through the bank slots as it is reused.) The descriptor for the new structure is used to gate **wptr**, before it can become **NEXT rptr**, to ensure the second invariant. When the loop completes, the bank will again be filled with empty structures, except for the hole corresponding to the final value of **X** at the final value of **rptr**. The setup and cleanup of the bank are represented by the obvious loop. Note that the cleanup need not be explicitly triggered by completion of the loop proper, but only by the final value of **rptr**, which gives the disposition of all entries in the bank.

Hoisting structure recycling so that a bank of structures can be used for many invocations of an s-loop is straightforward. The bank initialization is lifted into the setup of the outer loop and the deallocation into the cleanup. Successive invocations of the s-loop for a given bank are

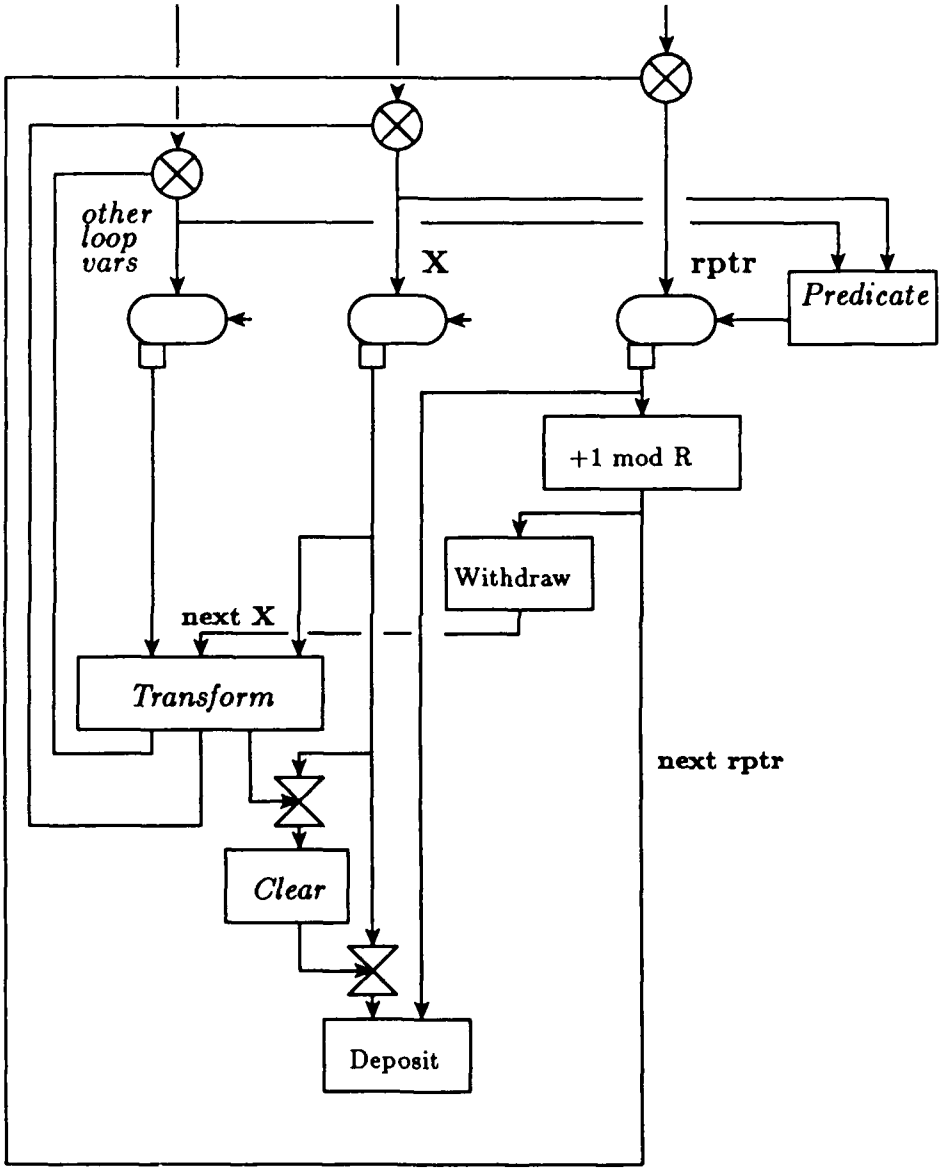


Figure A.13: s-Loop Schema

stitched together using the **rptr**. There is no need to explicitly test for completion of activity under one use of a bank before using it again, the availability of the **rptr** is sufficient. All other structures are either empty and in the bank or still in use and still withdrawn. The new invocation of the *s*-loop will wait if it tries to use a structure that is still withdrawn.

To avoid extraneous allocations we can initialize the bank with a special value and test for this value at the withdraw point. If the entry in the bank is the special value, the allocation is performed.

Appendix B

Estimating Token Storage

In this appendix, we develop a practical technique for analyzing the resource requirements of dataflow graphs. The space of legal configurations of a dataflow graph can be described by a collection of linear constraints, one per arc in the graph, and worst-case resource requirements can be obtained by maximizing a cost function representing the amount of resource used in a given configuration over this space. We focus first on token storage requirements, which requires the most detailed analysis, and then apply the techniques developed there to other resources.

To describe the space of legal configurations of a dataflow graph, we consider the relationships between adjacent nodes. The basic firing rule states that a node fires when a token is present on each of its input arcs, and upon firing it produces a token on each of its output arcs. Thus, if node i is connect to node j by a data dependence arc (i,j), then node j can only fire if i does.

Theorem 6 *Given an acyclic dataflow graph $G = (V,E)$ without conditionals let $\mathrm{F}(i)_c$ denote the number of times node $i \in V$ fires in a given configuration c. Then in any legal configuration $\mathrm{F}(j) \leq \mathrm{F}(i), \forall (i,j) \in E$ and any configuration satisfying these constraints is achievable in a legal execution sequence.*

The number of tokens on arc (i,j) in a given configuration is simply the difference in the number of times the two nodes fire, plus the number of tokens on the arc initially. Restating this in terms of nodes, the number of tokens in existence in configuration c is

$$\sum_{i \in V} outdegree_i - indegree_i,$$

plus the number of tokens in the graph initially.

B.1 Linear Constraint Systems to Model Resource Usage

For an acyclic graph without conditionals, we set up a linear program to determine the worst-case token storage requirement as follows. Augment the graph G = (V,E) with dummy source node v_0 and dummy sink node v_{n+1}, where $n = \|V\|$, connect v_0 to each input node in the graph, and connect each output node to v_{n+1}.

Linear Program 1 *Maximize* $\sum_{i \in V} c_i \mathrm{F}(i)$, *where* $c_i = outdegree_i - indegree_i$, *subject to:*

$$\mathrm{F}(j) - \mathrm{F}(i) \leq 0, \forall (i,j) \in E,$$
$$\mathrm{F}(0) - \mathrm{F}(n+1) \leq 1, \textit{and}$$
$$\mathrm{F}(i) \geq 0, \forall i \in V.$$

The second constraint is equivalent to introducing an additional arc from the dummy sink to the dummy source and placing a single token on this arc initially. For bounded loop graphs, this *feedback* arc is part of the graph initially, and we do not need to introduce the dummy source and sink nodes. For a cyclic code-block, take $G = (V, E)$ to be the loop portion of the graph, *i.e.*, excluding prelude and postlude. Let $M \subset V$ be the set of merge nodes, $\otimes$, at the top of the loop. Define $IT_{(i,j)}$ as the number of tokens on arc (i,j) in the initial configuration. Thus, for a k-bounded loop,

$$IT_{(i,j)} = \begin{cases} 1 & \text{if } i \in M, \\ k & \text{if } i = D_{-2}^{(k)} \text{ and } j = \text{L-gate, and} \\ 0 & \text{otherwise} \end{cases}$$

The linear program is as follows:

Linear Program 2 *Maximize* $\sum_{i \in V} c_i \mathrm{F}(i) + \sum_{(i,j) \in E} IT_{(i,j)}$, *where* $c_i = outdegree_i - indegree_i$, *subject to:*

$$\mathrm{F}(j) - \mathrm{F}(i) \leq IT_{(i,j)}, \forall (i,j) \in E, \textit{and}$$
$$\mathrm{F}(i) \geq 0, \forall i \in V.$$

This formulation in *standard form*; we put them in *canonical form* by introducing a slack variable $S_{(i,j)}$ for each $(i,j) \in E$, so that the Simplex algorithm can be used to find an optimal solution. The slack variable for an arc represents the number of tokens on the arc. The cost associated with a slack variable is zero. Thus, we have the following linear program in canonical form.

Linear Program 3 *Maximize* $\sum_{i \in V} c_i \mathrm{F}(i) + \sum_{(i,j) \in E} IT_{(i,j)}$, *where* $c_i = outdegree_i - indegree_i$, *subject to*

$$\mathrm{F}(j) - \mathrm{F}(i) + S_{(i,j)} = IT_{(i,j)}, \forall (i,j) \in E,$$
$$\mathrm{F}(i) \geq 0, \forall i \in V, \textit{and}$$
$$S_{(i,j)} \geq 0, \forall (i,j) \in E.$$

An initial basic feasible solution is obtained by taking the slack variables as the basis, with $S_{(i,j)} = IT_{(i,j)}$. The optimal solution can be obtained with the Simplex algorithm and will be integral, since the constraint matrix for the linear program above is *totally unimodular*[73]. However, the linear programs obtained in this way are very large. A code block with 300 nodes is not unusual, and, since we typically see 1.6 arcs per node, this implies a linear program with approximately 500 rows and 800 columns. Other techniques for solving these constraint systems are available, since the formulation is the dual of a min-cost flow problem, but these techniques are time-consuming on a problem this size, as well. Dramatic improvements can be made by reducing the graph to a much smaller one with equivalent worst-case resource behavior, however.

B.2 Reduced Constraint Graphs

Given a dataflow graph, we generate a smaller graph with the same worst-case resource behavior using two local transformations: *arc bundling* and *tree compression*, explained below. In effect, this reduction eliminates the trivial parts of the graph that could not be involved in the worst-case configuration, while preserving the essential dependencies. It typically results in a ten-fold reduction in each dimension of the constraint matrix, making the use of linear programming techniques to reach the final solution quite practical.

The linear program described above is slightly inaccurate, because a dataflow graph is a *multigraph*, allowing multiple arcs between a pair of nodes. In practice, this is not a problem, because we simply maintain a row for each. The second constraint expresses no new information, and the cost coefficients reflect the number of tokens that pass between the pair nodes correctly. We can eliminate some overhead by collapsing the redundant arcs into a single bundle, weighted by the number of arcs it contains.

Lemma 7 *(Arc bundling) Let resource constraint graph* $G = (V, E, W, IT)$, *where* $w_{(i,j)}$ *is the*

number of arcs in a bundle (i,j) and $IT_{(i,j)}$ is the number of tokens initially present on each of the arcs in bundle (i,j). Suppose G contains two bundles between a pair of nodes (i,j) with the same number of initial tokens, then the reduced graph G' obtained from G by collapsing these into a single bundle with weight equal to the sum of the pair has the same worst-case token storage as G.

Suppose $\delta F_{(i,j)} \equiv F(i) - F(j) > 0$ in the worst-case configuration. Then the number of tokens implied by the two bundles is simply the sum of their weights multiplied by $\delta F_{(i,j)}$. The same configuration is a solution in G', with the same cost. G' does not allow any configurations beyond those allowed for G, so this solution is optimal for G'. □

The resource constraint graph is obtained trivially from the dataflow graph by taking the weights to be unity. Bundling is important, not because redundant arcs are common in the dataflow graph, but because they occur frequently when tree compression is performed.

Lemma 8 *(Broadcast-Tree compression) Let $G = (V, E, W, IT)$ be a resource constraint graph and suppose node j is such that:*
(a) $Indegree_j = 1$, $(i,j) \in E$, $(j,k_1), ..., (j,k_n) \in E$,
(b) $w_{(i,j)} < \sum_{d=1}^{n} w_{(j,k_d)}$, and
(c) $IT_{(i,j)} = 0$.
Then, the graph G' obtained by eliminating node j and adding arcs $(i, k_d, w_{(j,k_d)}, IT_{(j,k_d)})$, for $d = 1..n$ has equivalent worst-case token storage.

Suppose G meets the conditions in the lemma, then Arc (i,j) cannot be occupied in the worst-case configuration; otherwise, a configuration with greater token storage would be obtained by "firing" node j. The only configurations allowed by G and not by G are those where tokens occupy this arc, so the worst-case configuration of G is obtained in G'. It is optimal for G' because all configurations in G are legal in G. The initial configuration of G is valid for G', by condition (c). □

Lemma 9 *(Reduction-Tree Compression) Let $G = (V, E, W, IT)$ be a resource constraint graph and suppose node j is such that:*
(a) $Outdegree_i = 1$, $(i,j) \in E$, $(k_1,i), ..., (k_n,i) \in E$,
(b) $w_{(i,j)} < \sum_{d=1}^{n} w_{(k_d,i)}$, and

(c) $IT_{(i,j)} = 0$.

Then, the graph G' *obtained by eliminating node* i *and adding arcs* $(k_d, j, w_{(k_d,i)}, IT_{(k_d,i)})$, *for* $d = 1..n$ *has equivalent worst-case token storage.*

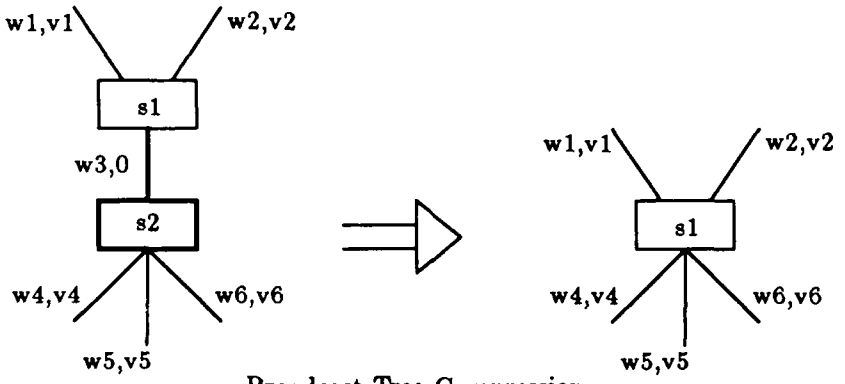
Same as above. □

These two transformations are shown graphically in Figure B.1. Each time a tree compression operation is performed, an opportunity for arc bundling may arise, since an arc may already exist from the parent of the eliminated node to one of its children. A single pass over the graph, in any order, suffices to compress all the trees originally embedded in the graph into their root. The complexity of this is more than linear in the size of the graph, because the branching factor may increase as the reductions are performed, but it is at most $O(\|V\|\|E\|)$. However, a single pass does not necessarily find all possible reductions, because new trees arise as nodes are eliminated. In practice, a single pass yields constraint systems small enough that the solution is obtained in less time than it takes to generate the graph. Figure B.2 shows the amount of compression achieved on the code-blocks in a large program.[1] On the average, the direct formulation of the linear program would yield a 126×210 tableau, but with tree compression this is reduced to 16×23.

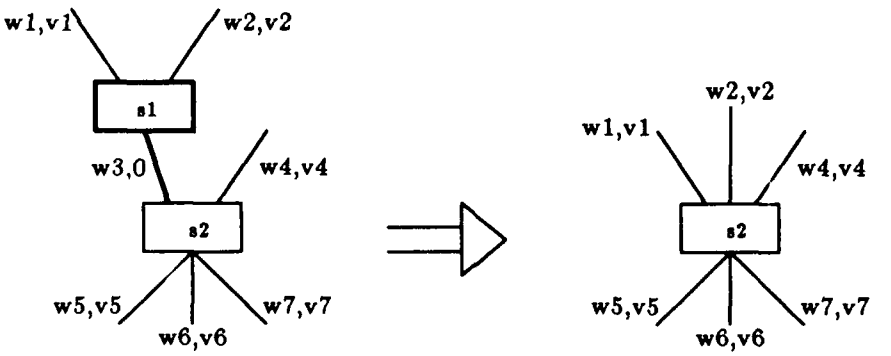
B.3 Conditionals

There are a number of ways to handle conditionals. The behavior of the conditional, *i.e.*, that data is passed only to one arm or the other, can be modeled by a collection of constraints. Unfortunately, the result constraint matrix is no longer unimodular. This means the optimal solution may not be integral, and, hence, is no longer a tight bound on the storage requirements. The brute force approach is to generate graphs for all possible settings of the conditionals and pick the worst. These two approaches can be combined in a branch-and-bound strategy. Alternatively, the conditional can be approximated by a graph which charges for both arms and ignores some of the dependencies, much as for the application construct discussed below. The numbers presented here were obtained by simply following the "true" side, and are optimistic in some cases.

[1] The program is an older version of the Simple code. It is expressed in a lower-level programming style than the version studied in Chapter 8 and includes fewer code-blocks.



Broadcast-Tree Compression



Reduction-Tree Compression

Figure B.1: Tree Compression Transformations

Code-Block Name	Nodes	Arcs	Rows	Cols
Simple	267	397	83	108
Make-Zero-Struct	65	87	3	5
Boundary-Node-Reflection	315	501	19	29
Boundary-Node-Reflection-0	65	91	11	16
Boundary-Node-Reflection-1	73	99	11	16
Reflect-Node	35	49	10	15
Boundary-Zone-Reflection	166	275	3	5
Boundary-Zone-Reflection-0	62	91	17	25
Boundary-Zone-Reflection-1	55	88	17	25
Velocity-Position	68	97	11	16
Velocity-Position-0	95	143	18	26
Hydro-Work-On-Boundary	90	137	3	5
Hydro-Work-On-Boundary-0	67	97	11	16
Hydro-Work-On-Boundary-1	87	117	11	16
Hydro-Energy	117	169	26	37
Hydro-Energy-0	230	345	43	60
Artificial-Viscosity	155	236	55	75
Conduction	421	632	3	5
Conduction-0	17	23	11	16
Conduction-1	63	93	11	16
Conduction-1-0	33	47	13	19
Conduction-1-1	18	24	11	16
Conduction-2	65	97	11	16
Conduction-2-0	18	24	11	16
Conduction-2-1	37	51	13	19
Conduction-3	68	99	29	41
Conduction-3-0	88	128	30	43
Conduction-4	44	62	11	16
Conduction-4-0	34	47	11	16
Conduction-5	35	48	11	16
Conduction-5-0	34	47	11	16
Boundary-Heat-Flow	57	79	3	5
Boundary-Heat-Flow-0	23	29	11	16
Boundary-Heat-Flow-1	23	29	11	16
Invert-Energy-Theta-Function	82	116	23	31
Polynomial	57	87	3	5
Table-Look-Up	28	37	3	5
Table-Look-Up-0	16	22	12	17
Table-Look-Up-1	16	22	12	17
AVERAGE	84.3	124.7	15.8	22.5

Figure B.2: Effects of Tree Compression and Bundling

B.4 Application Constructs

The application of a function or loop to a number of arguments in a dataflow program is represented in a dataflow machine graph as shown in Figure B.3.a. A Get-Context instruction, triggered by something in its environment, causes a *context* for the specified code-block to be allocated and passes a data item representing this context to instructions that send the argument values to the new context. The Form-Tag instruction forms a data item representing the tag of the first of a sequence of instructions that are to receive the results. This value is passed to the new context as argument zero. When all the results and a signal have been received from the new context, the context is released. The instructions that send arguments have no output arcs and the instructions that receive results have no input arcs, so special attention is required in forming the constraint graph. Code-block invocations are *non-strict*, so computation may begin before any of the arguments are received. In the extreme case, all the results are returned as soon as the context is allocated. We know only that all the arguments must be sent before the Release-Context can be enabled. We capture these minimal dependencies by introducing zero weight arcs as shown in Figure B.3.b. The receivers take an input from the node that sends the return tag. The other senders are connected to the Release-Context.

B.5 Tokens per Iteration

The token storage requirement for a k-bounded loop can be obtained by solving the linear program for $k = 2$, *i.e.*, one token on the arc controlling the gate and then performing sensitivity analysis on the final tableau[20]. Note, that the tree compression algorithm will not remove the edge between the $D_{-2}^{(k)}$ and the L-gate, because it has non-zero initial value. The row r corresponding to this arc initially contained 1 on the right-hand-side (RHS), an entry for the two nodes, and an entry for the slack variable S_r. The column corresponding to S_r initially contained all zeros, except the unit entry in row r. Thus, the cost coefficient C_r corresponding to S_r in the final tableau gives the cumulative scaling applied to row r in forming linear combinations with the cost row. C_r is the change in the objective function per unit increase in the initial RHS value of row r, *i.e.*, per unit increase in k. The range for which this scale factor is valid is determined by examining the remaining entries in the column for S_r. If any of them are negative, increasing the initial RHS of row r decreases the RHS of these rows in the final tableau.

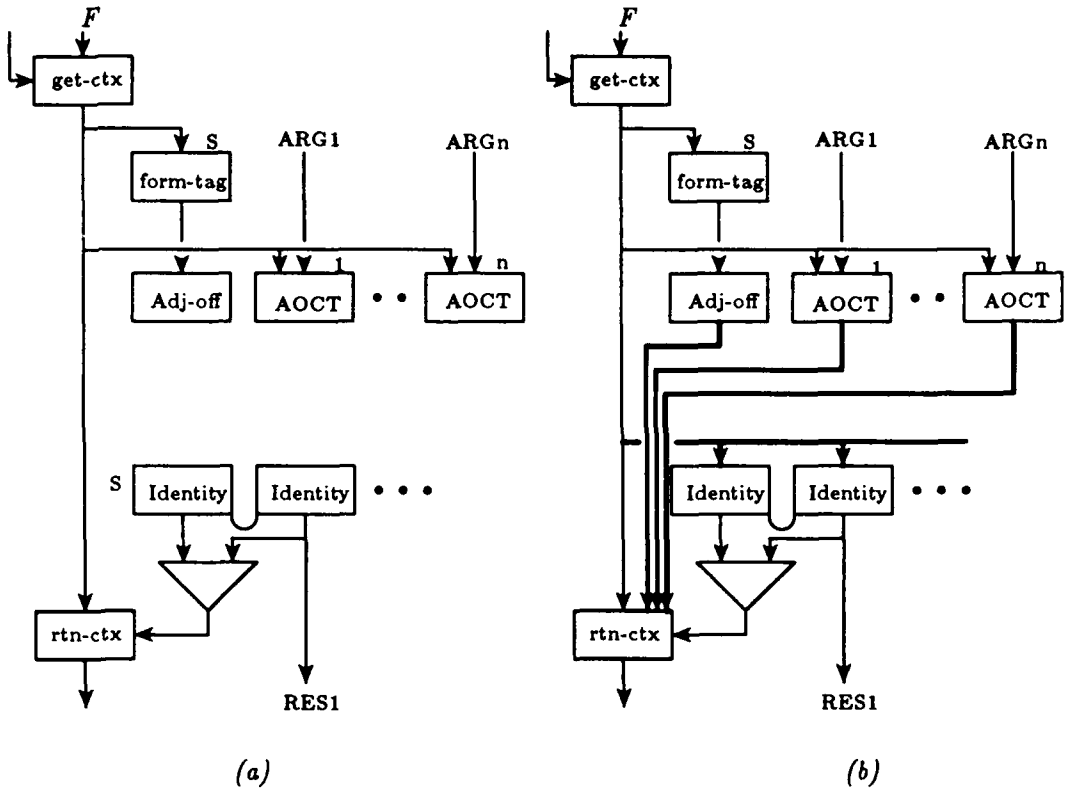


Figure B.3: Dataflow Machine Graph and Constraint Graph for Code-Block Application

The final basis is feasible as long as the RHS remains non-negative. A ratio test is applied to determine the column which leaves the basis, and a pivot is performed to produce a new basis. In practice, the range is almost always infinite, as the optimal solution established with $k = 2$ is a kind of steady state. It is possible, though difficult, to generate loops with a number of loop-variable dependence cycles of different lengths, so the behavior is not linear in k for a fixed range of values. For values of k greater than the number of loop variables, the behavior is linear.

Applying these techniques to the old Simple version, we get the token storage per code-block given in Figure B.4. For code-blocks containing loops, only the loop portion is considered, not the code around the loop.

B.6 Entire Programs

In the absence of general recursion, the token storage requirements of large portions of a program, or even entire programs, can be derived by first constructing the static invocation tree,

Code-Block Name	Tokens ($k = 2$)	per Iteration
Simple	472	235
Boundary-Node-Reflection	275	—
Boundary-Node-Reflection-0	89	44
Boundary-Node-Reflection-1	89	44
Reflect-Node	16	—
Boundary-Zone-Reflection	161	—
Boundary-Zone-Reflection-0	51	25
Boundary-Zone-Reflection-1	57	28
Velocity-Position	111	55
Velocity-Position-0	79	39
Hydro-Work-On-Boundary	100	—
Hydro-Work-On-Boundary-0	61	30
Hydro-Work-On-Boundary-1	61	30
Hydro-Energy	189	94
Hydro-Energy-0	301	150
Artificial-Viscosity	53	—
Conduction	361	—
Conduction-0	13	6
Conduction-1	93	46
Conduction-1-0	27	13
Conduction-1-1	13	6
Conduction-2	101	50
Conduction-2-0	13	6
Conduction-2-1	27	13
Conduction-3	98	48
Conduction-3-0	118	58
Conduction-4	59	29
Conduction-4-0	27	13
Conduction-5	47	23
Conduction-5-0	27	13
Boundary-Heat-Flow	56	—
Boundary-Heat-Flow-0	13	6
Boundary-Heat-Flow-1	13	6
Invert-Energy-Theta-Function	123	60
Polynomial	38	—
Table-Look-Up	20	—
Table-Look-Up-0	10	5
Table-Look-Up-1	10	5

Figure B.4: Worst-Case Tokens Storage for Code-Blocks in Old Simple

and solving the token storage requirements of each node, from the leaves upward. Obviously, this approach breaks down in the presence of general recursion or higher-order functions. Suppose the graph being analyzed invokes code-block F and the requirements of F are known. The arc between the Get-Context and the Return-Context for the application of F can be weighted by the worst-case requirement of F. This is conservative because it ignores all the constraints within F. Analysis of F could yield, in many cases, dependence relations between inputs and outputs, but what we would really need is the token storage associated with each combination of inputs present or not present and each combination of results produced or not. This is expensive to compute and difficult to summarize. If we know that F performs little useful computation without particular arguments being available, it would make sense to trigger the Get-Context by these arguments. This would make the constraint system technique more accurate and reduce the token storage requirements of the program. A difficulty remains, however, because the cost of the worst-case configuration of a code-block, in general, depends on the loop parameters of its subordinates.

In the absence of a complete dependence summary of an activation, weighting the return context arc is tantamount to charging for the worst-case of the subordinate code-blocks, in addition to the worst-case configuration of the code-block under analysis. Estimating resources under this more conservative strategy is straight-forward. We analyze each code-block in isolation, and then work up the call-tree augmenting each with the requirements of its subordinates. The various loop parameters can be carried along symbolically.

Bibliography

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*, volume 2: Compiling. Prentice-Hall, Englewood Cliffs, NJ, 1973.

[2] A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison-Wesley Pub. Co., Reading, Mass., 1977.

[3] A.V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, Reading, Massachusetts, 1974.

[4] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 233–246, June 1984.

[5] M. Amamiya, R. Hasegawa, O. Nakamura, and H. Mikami. A list-oriented data flow machine architecture. In *Proceedings of the National Computer Conference*, pages 143–151. AFIPS, 1982.

[6] Arvind and D. E. Culler. Managing resources in a parallel machine. In *Proceedings of IFIP TC-10 Working Conference on Fifth Generation Computer Architecture, Manchester, England.* North-Holland Publishing Company, July 1985.

[7] Arvind and D. E. Culler. Dataflow architectures. In *Annual Reviews in Computer Science*, volume 1, pages 225–253. Annual Reviews Inc., Palo Alto, CA, 1986. Reprinted in Dataflow and Reduction Architectures, S. S. Thakkar, editor, IEEE Computer Society Press, 1987.

[8] Arvind, D. E. Culler, and K. Ekanadham. The price of asynchronous parallelism: an analysis of dataflow architectures. In *Proceedings of CONPAR 88*, Univ. of Manchester, September 1988. British Computer Society — Parallel Processing Specialists. (also CSG Memo No. 278, MIT Laboratory for Computer Science).

[9] Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali, and R. E. Thomas. The Tagged Token Dataflow Architecture. Technical Report FLA memo, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, August 1983. Revised October, 1984.

[10] Arvind, D. E. Culler, and G. K. Maa. Assessing the benefits of fine-grain parallelism in dataflow programs. *The International Journal of Supercomputer Applications*, 2(3), November 1988.

[11] Arvind and K. Ekanadham. Future scientific programming on parallel machines. *The Journal of Parallel and Distributed Computing*, 5(5):460–493, October 1988.

[12] Arvind and K. P. Gostelow. A computer capable of exchanging processors for time. In B. Gilchrist, editor, *Proceedings of IFIP Congress 77*, pages 849–853, Toronto, Canada, August 1977. North-Holland.

[13] Arvind and K. P. Gostelow. The U-interpreter. *IEEE Computer*, 15(2), February 1982.

[14] Arvind, K. P. Gostelow, and W. Plouffe. An asynchronous programming language and computing machine. Technical Report 114a, University of California, Irvine, December 1978.

[15] Arvind and K.P. Gostelow. Some relationships between asynchronous interpreters of a dataflow language. In E. J. Neuhold, editor, *Proceedings of the IFIP WG2.2 Conference on Formal Description of Programming Languages*, St. Andrews, Canada, 1977. North-Holland.

[16] Arvind and R. A. Iannucci. Instruction set definition for a tagged-token dataflow machine. Technical Report 212, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, December 1981.

[17] Arvind and R. A. Iannucci. Two fundamental issues in multiprocessing. In *Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany*, June 1987.

[18] Arvind and R. S. Nikhil. Executing a program on the MIT Tagged-Token Dataflow Architecture. In *Proceedings of the PARLE Conference, Eindhoven, The Netherlands. (LNCS Volume 259)*. Springer-Verlag, June 1987. To appear in IEEE Transactions on Computers, revised June, 1988.

[19] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. Technical Report Computation Structures Group Memo 269, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, February 1987. (Also appears in *Proceedings of the Graph Reduction Workshop*, Santa Fe, NM. October 1986.).

[20] S. P. Bradley, A. C. Hax, and T. L. Magnanti. *Applied Mathematical Programming.* Addison-Wesley, Reading, Massachusetts, 1977.

[21] S. A. Brobst. Instruction scheduling and token storage requirements in a dataflow supercomputer. Master's thesis, Massachusetts Institute of Technology, Dept. of EECS, 77 Massachusetts Ave, Cambridge, MA, May 1986.

[22] V. J. Bush and J. R. Gurd. Transforming recursive functional programs for execution on parallel machines. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 350–367. Springer-Verlag, September 1985.

[23] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[24] D. Comte, N. Hifdi, and J. Syre. The data driven LAU multiprocessor system: Results and perspectives. In *Proceedings of IFIP Congress 80*, pages 175–180, Tokyo, Japan, October 1980.

[25] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The SIMPLE code. Technical Report UCID 17715, Lawrence Livermore Laboratory, February 1978.

[26] D. E. Culler. Resource management for the Tagged-Token Dataflow Architecture. Technical Report TR-332, MIT Laboratory for Computer Science, January 1985. (MS Thesis, Dept. of EECS, MIT).

[27] D. E. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 141–150, Hawaii, May 1988.

[28] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844, August 1986.

[29] R. Cytron. Limited processor scheduling of doacross loops. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 226–234, August 1987.

[30] R. Cytron and J. Ferrante. What's in a name? -or- The value of renaming for parallelism detection and storage allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, August 1987.

[31] G. Dahlquist and A. Bjork. *Numerical Methods.* Prentice-Hall, Englewood Cliffs, NJ, 1974.

[32] F. Darema-Rogers. Parallel applications development for shared memory systems. Technical Report RC 12229 (#54532), IBM T.J. Watson Research Center, Yorktown Heights, NY, August 1986.

[33] F. Darema-Rogers, A. Karp, and P. Teller. Applications survey reports — I. Technical Report RC 12743, IBM T.J.Watson Research Center, Hawthorne, New York, May 1987.

[34] J. R. B. Davies. Parallel loop constructs for multiprocessors. Master's thesis, University of Illinois, Urbana-Champaign, May 1981. Rep. No. UIUCDCS-R-81-1070.

[35] A. L. Davis. The architecture and system methodology of DDM1 : A recursively structured data driven machine. In *Proceedings of the 5th Annual International Symposium on Computer Architecture*, pages 210–215, April 1978.

[36] J. B. Dennis. First version of a data flow procedure language. In G. Goos and J. Hartmanis, editors, *Proc. Programming Symposium, Paris 1974 (Lecture Notes in Computer Science 19)*, New York, 1974. Spinger-Verlag. (Revised: MAC TM61, May 1975, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139).

[37] J. B. Dennis. Data flow supercomputers. *IEEE Computer*, pages 48–56, November 1980.

[38] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic dataflow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture*, page 126. IEEE, January 1975.

[39] J. J. Dongarra, *et al. Linpack User's Guide.* Society for Industrial and Applied Mathematics, 1980.

[40] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[41] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[42] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[43] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 11–16, Palo Alto, CA, July 1986. ACM.

[44] J. Gurd, W. Bohm, and Y. M. Teo. Performance issues in dataflow machines. In *Future Generation Computer Systems*, volume 3. North-Holland, 1987.

[45] J. Gurd, C.C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Communications of the Association for Computing Machinery*, 28(1):34–52, January 1985.

[46] J. Gurd and I. Watson. A multilayer data flow computer. In *Proceedings of the 1977 International Conference on Parallel Processing*, page 94, 1977.

[47] J. L. Gustafson. Reevaluating amdahl's law. *Communications of the Association for Computing Machinery*, 31(5):532–533, May 1988.

[48] J. L. Gustafson, G. R. Montry, and R. E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4), July 1988.

[49] W. Harrison. Compiler analysis of the value ranges for variables. Technical Report RC 5544, IBM T.J. Watson Research Center, Yorktown Heights, NY, July 1975.

[50] J. L. Hennessy. VLSI processor architectures. *IEEE Transactions on Computers*, C-33(12), December 1984.

[51] J. L. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[52] K. Hiraki, K. Nishida, S. Sekiguchi, and T. Shimada. Maintainence architecture and its LSI implementation of a dataflow computer with a large number of processors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 584–591, 1986.

[53] R. W. Hockney and C. R. Jesshope. *Parallel Computers.* Adam Hilger Ltd, Bristol, GB, 1981.

[54] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, NY, NY, 1984.

[55] R. A. Iannucci. A dataflow/von Neumann hybrid architecture. Technical Report TR-418, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, May 1988. (PhD Thesis, Dept. of EECS, MIT).

[56] R. G. Babb II, editor. *Programming Parallel Processors.* Addison-Wesley Pub. Co., Reading, Mass., 1988.

[57] K. Inoue, H. Seki, and H. Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM Transactions on Programming Languages and Systems*, 10(4):555–578, October 1988.

[58] H. F. Jordan. A guide to parallel computations and some Cray-1 experiences. In *Parallel Computations*, pages 1–50. Academic Press, 1982.

[59] R. H. Halstead Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Proceedings of the ACM Symposium on Lisp and Functional Languages*, August 1984.

[60] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[61] A. H. Karp. Programming for parallelism. *IEEE Computer*, 20(5):43–56, May 1987.

[62] R.M. Keller, G. Lindstrom, and S. Patil. A loosely-coupled applicative multi-processing system. In *Proceedings of the National Computer Conference*, volume 48, pages 613–622, New York, NY, June 1979.

[63] P. M. Kogge. *The Architecture of Pipelined Computers.* McGraw Hill, 1981.

[64] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of ACM Symposium on Principles of Programming Languages*, January 1981.

[65] M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers*, 37(9):1088–1098, September 1988.

[66] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, Inc, Englewood Cliffs, NJ, 1981.

[67] O. M. Lubeck. Supercomputer performance: The theory, practice, and results. In *Advances in Computers.* Academic Press, Orlando, FL, 1988.

[68] V. Markstein, J. Cocke, and P. Markstein. Optimization of range checking. In *Proceedings of SIGPLAN '82 Symposium on Compiler Construction*, pages 114–119, 1982.

[69] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, P. Hohensee, and I. Dobes. SISAL: streams and iteration in a single assignment language. Language Reference Manual, Version 1.2 M - 146, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[70] M. Montenyohl and M. Wand. Correct flow analysis in continuation semantics. In *Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 204–218, San Diego, CA, January 1988.

[71] R. S. Nikhil. Id (version 88.0) reference manual. Technical Report CSG Memo 284, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, March 1988.

[72] R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, May 1989. To appear.

[73] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[74] G. M. Papadopoulos. Implementation of a general purpose dataflow multiprocessor. Technical Report TR432, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, September 1988. (PhD Thesis, Dept. of EECS, MIT).

[75] D. A. Patterson. Reduced instruction set computers. *Communications of the Association for Computing Machinery*, 28(1):9–21, January 1985.

[76] C. D. Polychronopoulos. *On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems.* PhD thesis, University of Illinois, Urbana-Champaign, Center for Supercomputing Research and Decelopment, August 1986. CSRD Rpt. No. 595, UILU-ENG-86-8006.

[77] C. D. Polychronopoulos, D. J. Kuck, and D. A. Padua. Execution of parallel loops on parallel processor systems. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 519–527, 1986.

[78] C. A. Ruggiero. *Throttle Mechanisms for the Manchester Dataflow Machine.* PhD thesis, University of Manchester, Manchester M13 9PL, England, July 1987.

[79] C. A. Ruggiero and J. Sargeant. Hardware and software mechanisms for control of parallelism. Internal report, Department of Computer Science, University of Manchester, April 1985.

[80] J. Rumbaugh. A data flow multiprocessor. *IEEE Transactions on Computers*, C-26(2):138–146, February 1977.

[81] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* PhD thesis, Stanford University, Computer Systems Lab, Dept. of EE and CS, April 1987. CSL-TR-87-328.

[82] J. Savage. Parallel processing as a language design problem. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 221–224, Boston, Mass, June 1985.

[83] T. Shimada, K. Hiraki, and K. Nishida. An architecture of a data flow machine and its evaluation. In *Proceedings of CompCon 84*, pages 486–490. IEEE, 1984.

[84] N. Takahashi and M. Amamiya. A dataflow processor array system: Design and analysis. In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 243–250, Stockholm, Sweden, June 1983.

[85] K. R. Traub. A compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report TR-370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, August 1986. (MS Thesis, Dept. of EECS, MIT).

[86] K. R. Traub. Sequential implementation of lenient programming languages. Technical Report TR-417, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, September 1988. (PhD Thesis, Dept. of EECS, MIT).

[87] M. J. Wolfe. *Optimizing Supercompilers for Supercomputer.* PhD thesis, University of Illinois at Urbana-Champaign, 1982.

[88] M. J. Wolfe and U. Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Processing*, 16(2):137–178, April 1987.

[89] J. Young. Personal communication. M.I.T Computation Structures Group, 1989.

Vita

David Culler [redacted] He attended the University of California, San Diego before going to the University of California, Berkeley, where he received his A.B. in Mathematics in 1980. David worked at Lawrence Berkeley Laboratory and the National Magnetic Fusion Energy Computer Center before entering graduate school at the Massachusetts Institute of Technology to work with Prof. Arvind on dataflow architectures. He received his S.M. from M.I.T. in Electrical Engineering and Computer Science in January, 1985 and his Ph.D. in June, 1989. His work explores the interplay of resource management and parallel computer architecture.

OFFICIAL DISTRIBUTION LIST

DIRECTOR — 2 Copies
Information Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

OFFICE OF NAVAL RESEARCH — 2 Copies
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. Gary Koop, Code 433

DIRECTOR, CODE 2627 — 6 Copies
Naval Research Laboratory
Washington, DC 20375

DEFENSE TECHNICAL INFORMATION CENTER — 12 Copies
Cameron Station
Alexandria, VA 22314

NATIONAL SCIENCE FOUNDATION — 2 Copies
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

HEAD, CODE 38 — 1 Copy
Research Department
Naval Weapons Center
China Lake, CA 93555